Models and Algorithms for Hybrid Networks and Hybrid 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

Kristian Hinnenthal

PADERBORN UNIVERSITY

Reviewers:

- Christian Scheideler,
  Paderborn University
- Friedhelm Meyer auf der Heide,
  Paderborn University
- Irina Kostitsyna,
  Eindhoven University of Technology

To Linus, my beloved son.

Abstract

This dissertation is devoted to the study of two different hybrid distributed systems, hybrid networks and hybrid programmable matter.

Hybrid networks are communication networks in which the nodes possess different modes of communication. For example, modern mobile phones can typically communicate both via Bluetooth ad hoc connections as well as the cellular network, but applications rarely leverage both. To study hybrid networks, in the first part of this thesis we establish a theoretical model in which nodes have two different communication modes, a local and a global mode. Whereas the local mode captures characteristics of a limited-range fixed network, such as an ad hoc network, the global mode models the nodes' ability to use a shared infrastructure such as the cellular network. We explore the capabilities and limitations of hybrid networks under different communication restrictions and present algorithms for various problems. As a main focus of the first part, we study the computation of graph problems, and shortest paths in particular, where the input graph is given by the local network.

The second part of this dissertation revolves around hybrid programmable matter. Hybrid programmable matter refers to a system of minute robots that operate on a set of tiles, and that can lift and place tiles to alter the structure. The robots are envisioned to act autonomously and without any global information, only based on a simple internal program. Therefore, from the outside, the system behaves as a self-transforming substance. We study two different problems in a simple model for hybrid programmable matter, shape formation and shape recognition. The main focus of our work lies in exploring the power of a single robot, and we lay some foundations to leverage multiple robots that act coordinately.

Zusammenfassung

Diese Dissertation behandelt zwei Typen von hybriden verteilten Systemen, hybride Netzwerke und hybride programmierbare Materie.

Hybride Netzwerke bezeichnen Kommunikationsnetzwerke in denen Knoten verschiedene Kommunikationsmodi besitzen. Moderne Mobiltelefone können beispielsweise sowohl Bluetooth-Verbindungen aufbauen, als auch über das zelluläre Netzwerk kommunizieren. Praktische Anwendungen, die beide Kommunikationsmodi ausnutzen, sind dennoch selten. Im ersten Teil dieser Dissertation stellen wir ein theoretisches Modell für hybride Netzwerke mit zwei verschiedenen Kommunikationsmodi vor, einem lokalen und einem globalen Modus. Während der lokale Modus die Eigenschaften von vorgegebenen Netzwerken begrenzter Reichweite, wie ad hoc Netzwerken, erfasst, beschreibt der globale Modus die Fähigkeit der Knoten über eine geteilte Infrastruktur, wie dem zellulären Netzwerk, zu kommunizieren. Wir untersuchen die Möglichkeiten und Grenzen eines solchen hybriden Netzwerkes für unterschiedliche Kommunikationsbeschränkungen und präsentieren Algorithmen für verschiedene Probleme. Einer der Schwerpunkte des ersten Teils dieser Dissertation liegt im Studium von Graphproblemen wie der Berechnung kürzester Wege im Graphen der lokalen Kanten.

Der zweite Teil dieser Dissertation behandelt hybride programmierbare Materie. Hybride programmierbare Materie setzt sich zusammen aus aktiven Elementen, den Robotern, die sich auf passiven Elementen, den Kacheln, bewegen und diese verschieben können. Die Roboter agieren dabei autonom und ohne globale Informationen, nur basierend auf einem einfachen gespeicherten Programm, und bilden gemeinsam mit den Kacheln eine Art programmierbare Substanz. Wir untersuchen zwei verschiedene Probleme in einem einfachen Modell für hybride programmierbare Materie, das Shape Formation Problem sowie das Shape Recognition Problem. Der Fokus unserer Arbeit liegt in der Erforschung der Mächtigkeit eines einzelnen Roboters. Darüber hinaus präsentieren wir grundlegende Erkenntnisse für die Forschung in Systemen, die aus einer Vielzahl von Robotern und Kacheln bestehen.

Acknowledgments

I am most grateful for the support and advice of Christian Scheideler, under and with whom I had the honor to work for the last four years. Thank you for answering every single question I raised and always taking time for discussions whenever I asked. Your professional and personal support was and still is invaluable to me, and allowed me to thrive in and enjoy the research we did.

I would also like to thank my co-authors and colleagues, with whom I had countless interesting discussions and who never ceased to amaze me with their brilliant ideas. I particularly thank Robert Gmyr, Dorian Rudolph, Thorsten Götte, Michael Feldmann, and Philipp Schneider, who worked most closely together with me and whose contributions can hardly be overstated. Thank you as well to all colleagues who carefully proof-read my work, and to everyone really who had to put up with me. Thanks for being patient and always supporting.

Lastly, I would like to thank my beloved wife Miriam, who carried me throughout the last decade and who never got tired of hearing me talk about my work. Thanks for always supporting me no matter what. Your love means the world to me.

Contents

1 Introduction

This thesis explores two different kinds of *hybrid distributed systems.* In the first part of this thesis, we study *hybrid networks.* In contrast to classical communication networks, in hybrid networks nodes can communicate using *multiple* modes of communication with different characteristics. The second hybrid system we investigate is *hybrid programmable matter.* Such a system consists of small *active* elements that act on a set of *passive* tiles. The active elements are envisioned as small autonomous robots with the computational capabilities of a finite automaton that can be used to explore and manipulate any given tile structure. We first give some motivation and a more detailed explanation of these two concepts and then provide an overview of the results of this dissertation.

Hybrid Networks The general idea of *hybrid communication* is to leverage multiple, but fundamentally different, communication modes instead of relying on a single mode only. As a matter of fact, virtually all communication we perform daily is hybrid. Spoken language, as an example, is often accompanied by some form of *nonverbal* communication such as gestures or body language. Whereas the former can be used to directly exchange information, ideas, or desires, the latter often serves to augment or contextualize the given information. For example, the reputation of a fisherman clearly depends on how long he can stretch his arms when describing his latest catch. These two modes of communication exhibit very different characteristics, which we intuitively take into account when speaking to each other. Speech can, for example, only be heard up to a limited distance, but may also propagate around corners. Gestures, on the other hand, may be discernible over longer range, but only help if the gesticulating person can actually be seen.

Although modern computer devices typically possess a multitude of communication modes, applications that actually make use of a combination of different modes are seldom. As an example, modern cell phones can, in principle, make use of Bluetooth or Wi-Fi ad hoc connections as well as the cellular network infrastructure. However, a typical messenger application will not dynamically decide upon the appropriate communication mode to carry out a message transmission, depending on the location of the recipient, for instance. In fact, the 5G communication standard, which has only recently reached wide-spread usage, is the first mobile standard that is designed to utilize device-to-device communication in addition to cellular networks [KS18]. A promising application of such *hybrid networks* formed using 5G can be found in vehicle-to-vehicle communication [SM15].

As another example of hybrid networks that are already used in practice, *hybrid data center networks* [FS19] combine traditional electronic packet switching with dedicated connections between servers. These dedicated connections can be formed using optical switches (e.g., [Far+10; Wan+10]), wireless antennas (e.g., [Hal+11; Zho+12]), or laser connections (e.g., [Ham+14]). Combining the different communication modes helps to deliver scalable throughput, or to reduce a system's complexity, cost, or power consumption. Other examples of hybrid communication include *dynamic multipoint VPNs*, which leverage a combination of dedicated leased lines with standard, best-effort VPN connections [RS11], or *hybrid WANs*, which incorporate both the internal communication infrastructure of a company, for example, as well as communication via the Internet [Tel+18].

Although the utility of hybrid networks has been proven in practice, rigorous theoretical research is still in its infancy. In the first part of this thesis, we try to advance in this direction. First, we establish a general theoretical model for hybrid networks that leverage *two* different communication modes, a *local* mode and a *global* mode. Using the local mode, nodes can send messages to their neighbors in a fixed communication graph that we call the *local network*. On the other hand, the global mode allows nodes to, in principle, send messages to *any* other node in a potentially dynamic *global network*. As the topology of the local network prescribes which nodes can communicate directly, it may allow nodes to contact many nodes at the same time. By contrast, the communication graph of the global network, albeit being much more flexible, only allows the nodes to communicate with very few other nodes at the same time. The very distinct capabilities and limitations of the two network types capture characteristic properties of many interesting hybrid networks: The local network can be used to model a fixed communication infrastructure such as ad hoc connections of mobile devices or the physical infrastructure of a data center, whereas the global network captures the ability to establish dedicated connections, for example, via the Internet or laser connections between servers. Based on this model, we study multiple aspects of hybrid networks in this thesis.

Hybrid Programmable Matter The second part of the dissertation revolves around a different kind of hybrid distributed systems, namely *hybrid programmable matter*. Programmable matter refers to a collection of minute entities that act in coordination to achieve some desired goal. We envision the entities to act without external control, only according to their internal logic and based on their local information, hence the term *programmable*. From the outside, single entities are indistinguishable and almost imperceptible, making the system appear as *matter*. The number of potential practical applications of such systems is incalculable. For example, programmable matter may be used in minimal invasive surgery, where it could seal wounds, build stents, or capture and disarm cancerous cells. It can also be helpful in places that are too arduous or dangerous for humans to reach, such as the outer space, or that are inaccessible to conventional robotic systems, such as in medical applications.

There exist many models for programmable matter that assume different capabilities and limitations of a system. The models can generally be divided into two classes, namely models that assume *active* or *passive* entities. An example for active programmable matter is the *Amoebot model* [Der+14], in which small robots with the computational capabilities of finite automata move by performing contractions and expansions. Many interesting problems for programmable matter have been investigated under this model, for example *shape formation* [Der+16; Di +20], *coating* [Der+17; Day+18], *hull formation* [Day+20], and *leader election* [Day+17]. The Amoebot model, as well as similar active models [Woo+13; Hur+15], requires *each* entity to perform computations and control its own movement. Whereas this allows the system to solve complex tasks, realizing the entities may be difficult and costly.

An approach to *passive* programmable matter is to use *DNA tiles* [Pat14]. The tiles are constructed in nanometer scale from DNA, and can bind to each other in order to assemble larger structures. One of the most prominent models for DNA tile self-assembly is the *abstract tile-assembly model* (aTAM) [RW00]. Here, the way in which the tiles attach to each other is defined by different types of *glues* rather than deliberate movements from one position to another. Additional changes to the structure have to be enforced externally, for example by changing the temperature or exposing the structure to certain kinds of radiation.

In this thesis, we investigate a *hybrid model* for programmable matter, in which we are given a set of *passive* tiles that are uniform and stateless, and a limited number of *active* robots. The robots, which only have the computational power of finite automata, move on the structure of tiles. By transporting tiles from one position to another, they are able to rearrange the structure, for example, to form some desired shape. Compared to the DNA tile-based approach, in our model, all tiles are of the same type and movements are exclusively performed by the robots. Furthermore, in contrast to the approaches based entirely on active elements, we believe that many problems can be solved in our hybrid model using only a few active elements. We specifically investigate two problems for hybrid programmable matter, namely *shape formation*, and *shape recognition*, where we mostly focus on the single-robot case.

Although the complexity of our model is very restricted, actually realizing such a system is currently still a challenging task. A promising candidate for a potential realization of hybrid programmable matter lies in *DNA nanomachines*, for which there has been significant progress in recent years. For example, nanomachines have been demonstrated to be able to act as the head of a finite automaton on an input tape [RS09], to walk on a one- or two-dimensional surface [Lun+10; OSS09; Wic+12], and to transport cargo [Thu+17; SP04; WEW12]. We therefore believe that at some point it may be feasible to build nanomachines with the capabilities assumed in this dissertation.

Thesis Overview

In the following, we outline the structure of this thesis, summarize the main results of each chapter, and present the underlying publications. The hybrid network part of this thesis is prefaced with a prologue in Chapter 2, which describes and discusses the hybrid network model that underlies the four subsequent chapters. Additionally, the chapter contains an overview of the definitions and notations used throughout the first part of this thesis, and provides a summary of recurring concepts and problems.

Chapter 3: Fast Construction of Overlay Networks We begin this thesis by investigating the case in which the global network is not initially given, but needs to be *constructed*. More precisely, we assume that the local network forms some connected graph G, and each node initially only knows its neighbors in G. For the nodes to effectively utilize global communication, we first need to establish a suitable structure of global edges. This problem is also known as the *Overlay Construction Problem*: Given some connected graph, the goal is to transform it into some suitable overlay network such as a balanced tree, a butterfly network, or a sorted ring. To solve the Overlay Construction Problem, the nodes can introduce neighbors to one another by sending messages that contain node identifiers. Thereby, additional global edges can be established. However, the nodes are only allowed to communicate a polylogarithmic number of bits in each round using global edges, which makes the naive approach of introducing all neighbors to each other until the network forms a clique infeasible.

Specifically, we want to construct a tree that has constant degree and height $\mathcal{O}(\log n)$. It is easy to see that it takes time $\Omega(\log n)$ to construct such a tree starting from a line, even if unbounded communication was allowed. In this thesis, we present an algorithm that comes close to this lower bound and only requires $\mathcal{O}(\log^{3/2} n)$ rounds. At the heart of our algorithm lies a deterministic strategy to group and merge large components of nodes, but we make use of randomized load-balancing techniques to comply with the communication constraints. The chapter is based on the following publication, which we adapt to our hybrid network model and improve to also compute a spanning tree of G as a byproduct.

> T. Götte, K. Hinnenthal, and C. Scheideler. “Faster Construction of Overlay Networks”. In: *International Colloquium on Structural Information and Communication Complexity (SIROCCO)*. 2019, pp. 262–276 [GHS19]

Chapter 4: Distributed Computation with Node Capacities After discussing the problem of constructing a suitable global network in Chapter 3, the focus of Chapter 4 is on the difficulty of distributed computation with *node capacities*. In contrast to the local network, we impose a communication bound for the global network at each *node* rather than limiting the capacity for each edge. Therefore, although each node can in principle communicate with any other node using the global network, the total amount of global communication that a node can perform in a fixed amount of time is highly limited. To study the effect of nodes having limited communication capacity on the complexity of distributed computations, we introduce the *node-capacitated clique* (NCC) model. The model fully abstracts from the problem of constructing a global network and simply assumes that the nodes are arranged as a clique. In each round, every node can exchange messages of $\mathcal{O}(\log n)$ bits with at most $\mathcal{O}(\log n)$ other nodes. When solving a graph problem, the input graph G is defined on the same set of n nodes, where each node knows which other nodes are its neighbors in G.

We present distributed algorithms for the *Minimum Spanning Tree*, *BFS Tree*, *Single-Source Shortest Paths* (SSSP), *Maximal Independent Set* (MIS), *Maximal Matching*, and *Vertex Coloring* problems. We show that even with only $\mathcal{O}(\log n)$ concurrent interactions per node, the MST problem can still be solved in polylogarithmic time. In all other cases, the runtime of our algorithm depends linearly on the *arboricity* of G, which is the minimum number of forests into which the edges of G can be partitioned. For many important graph classes such as planar graphs, the arboricity is a constant. The chapter is an extension of the following publication.

> J. Augustine, M. Ghaffari, R. Gmyr, K. Hinnenthal, F. Kuhn, J. Li, and C. Scheideler. "Distributed Computation in Node-Capacitated Networks". In: *Proceedings of the 31st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 2019, pp. 69–79 [Aug+19]

Chapter 5: Shortest Paths in Sparse Hybrid Networks The previous chapter demonstrates that some problems are difficult to solve if each node can *only* perform small global communication. In particular, shortest path problems such as SSSP or computing the diameter seem to be inherently hard. Therefore, in Chapter 5 we return to our assumptions made in Chapter 3 and allow small local communication.

More precisely, we show how to compute SSSP and the diameter very efficiently in *sparse graphs.* For these problems, we present exact randomized $\mathcal{O}(\log n)$ time algorithms for cactus graphs (i.e., graphs in which each edge is contained in at most one cycle), and 3-approximations for graphs that have at most $n + \mathcal{O}(n^{1/3})$ edges and arboricity $\mathcal{O}(\log n)$. As intermediate steps, we describe deterministic $\mathcal{O}(\log n)$ time solutions for these problems in trees, cycles, and pseudotrees (i.e., trees that have at most one cycle). Our algorithms heavily rely on the tools established in the previous chapters and techniques known from parallel computing. The chapter is based on the following publication.

> M. Feldmann, K. Hinnenthal, and C. Scheideler. "Fast Hybrid Network Algorithms for Shortest Paths in Sparse Graphs". In: *Proceedings of the 24th International Conference on Principles of Distributed Systems (OPODIS).* 2020, 31:1–31:16 [FHS20]

Chapter 6: Shortest Paths in General Hybrid Networks To explore the full power of hybrid networks, in Chapter 6 we study a hybrid model that combines the NCC as the global network with a local network that allows *unbounded* communication. In contrast to the previous section, in which we focused our attention on sparse graphs, the increased potential of the model allows us to solve shortest path problems in *general graphs.* More precisely, we present two algorithms for the SSSP Problem. The first algorithm solves SSSP exactly in time $\widetilde{\mathcal{O}}(\sqrt{\mathsf{SPD}})$, where SPD is the so-called *shortest-path diameter* of G, and the tilde-notation hides all factors that are polylogarithmic in n. It exploits the capability of the nodes to communicate large subgraphs to their neighbors, and makes heavy use of the techniques established in previous sections. Our second contribution is an algorithm that approximates SSSP up to a multiplicative $(1/\varepsilon)^{\mathcal{O}(1/\varepsilon)}$-factor in time $\widetilde{\mathcal{O}}(n^{\varepsilon})$. The algorithm is based on building a hierarchy of *skeleton spanners*, which are sparse graphs that are constructed in a recursive fashion using the global network. The chapter is based upon the following publication.

J. Augustine, K. Hinnenthal, F. Kuhn, C. Scheideler, and P. Schneider. "Shortest Paths in a Hybrid Network Model". In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* 2020, pp. 1280–1299 [Aug+20b]

The original publication also contains algorithms for *All-Pairs Shortest Paths* (APSP), which requires techniques beyond the scope of this thesis. For comprehensiveness, we provide an overview of the omitted results and the additional techniques in the chapter.

Chapter 7: Shape Formation in Hybrid Programmable Matter We begin our study of hybrid programmable matter with the problem of *shape formation.* As a first step towards developing a general framework for these problems, we consider the problem of rearranging a connected set of hexagonal tiles by a *single* deterministic finite automaton. After investigating some limitations of a single-robot system, we show that a feasible approach to build a particular shape is to first rearrange the tiles into an intermediate structure by performing very simple tile movements. We introduce three types of such intermediate structures, each having certain advantages and disadvantages. Each of these structures can be built in asymptotically optimal $\mathcal{O}(n^2)$ rounds, where n is the number of tiles. As a proof of concept, we give an algorithm to reconfigure a set of tiles into an equilateral triangle through one of the intermediate structures. Finally, we experimentally show that the algorithm for building the simplest of the three intermediate structures can be modified to be executed by multiple robots in a distributed manner, achieving an almost linear speedup in the case where the number of robots is reasonably small. We further explain how the algorithm can be used to construct a triangle distributedly. The chapter is based on the following publication.

R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, C. Scheideler, and T. Strothmann. "Forming Tile Shapes with Simple Robots". In: *Proceedings of DNA Computing and Molecular Programming (DNA).* 2018, pp. 122–138 [Gmy+18c]

An extended version of the paper has also been published in the *Natural Computing* journal.

R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, C. Scheideler, and T. Strothmann. "Forming tile shapes with simple robots". In: *Natural Computing* 19.2 (2020), pp. 375–390 [Gmy+20]

Chapter 8: Shape Recognition in Hybrid Programmable Matter In the final chapter of our thesis, we investigate the problem of detecting the geometric shape of a given structure by a single robot. In particular, we consider the question of recognizing whether the tiles are assembled into a parallelogram whose longer side has length $\ell = f(h)$ for a given function $f(\cdot)$, where h is the length of the shorter side. To determine the computational power of the finite-state automaton robot, we identify functions that can or cannot be decided when the robot is given a certain number of pebbles. We show that the robot can decide whether $\ell = ah + b$ for constant integers a and b without any pebbles, but cannot detect whether $\ell = f(h)$ for any function $f(x) = \omega(x)$. For a robot with a single pebble, we present an algorithm to decide whether $\ell = p(h)$ for a given polynomial $p(\cdot)$ of constant degree. We contrast this result by showing that, for every constant s, any function $f(x) = \omega(x^{6s+2})$ cannot be decided by a robot with s states and a single pebble. We further present exponential functions that can be decided using two pebbles. Finally, we describe a family of functions $f_k(\cdot)$ for which the robot needs more than k pebbles to decide whether $\ell = f_k(h)$. The chapter is based on the following publication.

R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, and C. Scheideler. "Shape Recognition by a Finite Automaton Robot". In: *Proceedings of the 43rd International Symposium on Mathematical Foundations of Computer Science (MFCS).* 2018, 52:1–52:15 [Gmy+18a]

Part I.

Hybrid Networks

2 Prologue

The first four publications of this thesis, upon which Chapters 3 to 6 are based, can all be generalized under the same hybrid network model. Moreover, the chapters mostly rely on the same set of definitions and often employ similar techniques, or solve similar tasks under varying assumptions. This preliminary chapter contains a general model definition that combines the assumptions of all chapters concisely and gives an overview of the specific model assumptions of each chapter. In Section 2.2, we then provide a basic set of definitions and notations. We conclude this chapter in Section 2.3 with a summary of some of the recurring problems of the first part of this thesis.

2.1. Model

Each chapter of the first part of this thesis focuses on a different aspect of hybrid networks and is therefore based on specific model assumptions. The models can be generalized under the *generic hybrid network model*, which we formally introduce in this section. Our model combines the approach of Augustine et al. [Aug+20b] to parameterize the communication restrictions for the local and global network with the idea to view the global network as a dynamic overlay network [Gmy+17a].

To the best of our knowledge, no models comparable to our hybrid network model have been studied before the above-mentioned publications. Therefore, we forego a summary of models that are somewhat related to ours at this point, and only compare ourselves to literature whenever appropriate. In addition to that, each chapter contains an overview of the specific related work of that chapter.

Local and Global Network We are given a fixed set V of n nodes, where we assume the nodes to know an upper bound $\hat{n} \geq n$ on n that is polynomial in n. The nodes are connected via two kinds of edges: *local* edges and *global* edges. The local edges form a *fixed* local network. Formally, we represent the local network by an undirected, weighted graph $G = (V, E, w)$, where the edge weights are given by $w : E \to \{1, \dots, \mathcal{W}\} \subseteq \mathbb{N}$ for some $\mathcal{W}$ that is at most polynomial in n. Thus, every weight and length of any shortest path can be represented using $\mathcal{O}(\log n)$ bits. The graph G is said to be *unweighted* if $w : E \to \{1\}$.

We assume the standard synchronous message passing model, where time is divided into synchronous *rounds*. Each node $u \in V$ has a unique *identifier* $\text{id}(u)$, which is a bit string of length $\mathcal{O}(\log n)$. 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 global edges in round i as $D_i = \{(u, v) \mid u \in V \text{ and } v \in D_i(u)\}$. In contrast to the local network, the global network is therefore *dynamic* and may change over time.

Communication Capacities In each round every node can send distinct messages of size $\mathcal{O}(\log n)$ to other nodes using both its local and global edges. Since $\mathrm{id}(u) = \mathcal{O}(\log n)$ for all $u \in V$, each message can contain a constant number of node identifiers. If u sends $\mathrm{id}(w)$ to v in round i, then $(v, w) \in D_{i+1}$. Therefore, additional global edges can be set up by sending node identifiers. However, the number of messages that can be sent over the local and global network is restricted by parameters λ and γ: The *local capacity* λ is the maximum number of messages that can be sent over each local edge in a round, and the *global capacity* γ is the maximum number of messages any node can send and receive via global edges in a round. When in some round more than λ (or γ) messages are sent over an edge (or to a node, respectively), we assume that an adversary delivers an arbitrary subset of these messages and drops the other messages. All algorithms in this thesis ensure, either deterministically or with high probability, that a node never sends or receives too many messages.

Note that whereas λ imposes a bound on the number of messages that can be sent over each *edge*, γ effectively restricts the amount of global communication at each *node*. This modeling choice is motivated by the idea that local communication rather relates to *physical* networks, where an edge corresponds to a physical connection (e.g., cable- or ad hoc networks), whereas global communication primarily captures aspects of *logical* networks that are formed as an overlay on top of some shared physical infrastructure. In overlay networks, however, the amount of information that a node can send out in a single round does not scale linearly with the number of its incident edges. Therefore, it is more reasonable to impose a bound on the amount of information that a *node* can send and receive in one round, rather than imposing a bound on the amount of information that can be sent along each of its incident *edges*.

Model Discussion For appropriate choices of λ and γ, our model captures various established network models. For $\gamma = 0$, the nodes can only use the local edges as in classical fixed communication networks. If in this case we choose $\lambda = \infty$, our model corresponds to the LOCAL model of distributed computing. The related CONGEST model [Pel00], which is used to study the impact of edge capacities, is obtained by setting $\lambda = \mathcal{O}(1)$. If additionally G is a clique, then our model corresponds to the *congested clique*, which has initially been proposed as a simple model for *overlay networks* [Lot+05]. However, the congested clique allows each node to be in contact with up to $\Theta(n)$ other nodes at the same time. As we argue in Chapter 4, this issue seems to severely limit the utility of the congested clique as a suitable model for overlay networks.

As a more realistic communication bound that ensures scalability in overlay networks, many recent publications assume that each node can only send and receive a *polylogarithmic* number of bits in each round [GHS19; GVS19; AS18; Gmy+17a; DGS16]. Such networks are captured by the global network in our model for $\lambda = 0$ and $\gamma = \widetilde{\mathcal{O}}(1)$ (recall that $\widetilde{\mathcal{O}}(\cdot)$ hides factors polylogarithmic in n). The *node-capacitated clique* (NCC) model we propose in Chapter 4 is a simplified model for overlay networks. It allows each node to only send and receive at most $\mathcal{O}(\log n)$ messages, which is small enough to ensure scalability without necessitating overly complicated techniques. Furthermore, it abstracts away the necessity to explicitly *establish* overlay edges by simply assuming that the network forms a clique. Therefore, if we additionally assume that every node knows the identifiers of all other nodes, the generic hybrid model captures the NCC with $\lambda = 0$ and $\gamma = \mathcal{O}(\log n)$.

As an intermediate model between these two types of overlay network models, Augustine et al. [Aug+20a] proposed to study the so-called NCC_0 model. Instead of all nodes forming a clique, the model assumes that the nodes initially only know their neighbors in some input graph H. However, the nodes are still restricted to only communicate $\mathcal{O}(\log n)$ messages in each round per node. Clearly, our hybrid model incorporates the NCC_0 for $\lambda = 0$ and $\gamma = \mathcal{O}(\log n)$ if the initial knowledge graph of the global network corresponds to H.

Besides the above-mentioned models, our hybrid network model is also somewhat related to the *multimedia network* model of Afek et al. [Afe+90], who might have been the first to propose theoretical research with multiple communication modes. They consider a combination of a point-to-point message passing network akin to our local network and a multiaccess channel, which nodes can use to broadcast messages to all other nodes. This second mode is related to our global network in that it allows nodes to contact any other node. However, since the multiaccess channel allows nodes to contact *all* other nodes, and, on the other hand, only permits a *single* node to use the channel at a time, the two models are hardly comparable. This observation is, for example, supported by the fact that computing simple aggregate functions such as sums requires time $\Omega(n)$ in the multimedia model [Afe+90], whereas it only requires $\mathcal{O}(\log n/\log\log n)$ rounds in the NCC (see Chapter 4).

Model Variants in this Thesis Each communication model underlying a chapter of the first part of this thesis can be seen as an instance of our generic hybrid model. In Chapter 3, we assume a very restricted setting in which $\lambda = \mathcal{O}(1)$, $\gamma = \mathcal{O}(\log n)$, and each node initially only knows its neighbors in G, i.e., $D_1(u) = \{v \mid \{u, v\} \in E\}$ for all $u \in V$. Since this network model can be regarded as a combination of the $\mathsf{CONGEST}$ model for the local network and the NCC_0 for the global network, we will refer to it as the $\mathsf{CONGEST}+\mathsf{NCC}_0$ model. As mentioned above, the NCC we study in Chapter 4 is an instance of our generic model as well.

When we turn our attention to shortest path problems in hybrid networks in Chapter 5, we consider the hybrid model of Chapter 3, but assume that the global network forms a clique. Since this model corresponds to a combination of the $\mathsf{CONGEST}$ and the NCC model, we refer to it as the $\mathsf{CONGEST}+\mathsf{NCC}$ model. This simplification can be justified by our insights on overlay construction in Chapter 3, and allows us to use our techniques from Chapter 4 mostly as a black box. In fact, as we argue in Chapter 5, all algorithms of the chapter can be adapted to also work in the $\mathsf{CONGEST}+\mathsf{NCC}_0$ model with little effort.

While we focus on *sparse* graphs in Chapter 5, we finally investigate shortest path problems in *general* graphs in Chapter 6. However, since dense graphs prove to be much more difficult with limited local capacity, we allow the nodes to perform an arbitrary amount of local communication. Consequently, we refer to this model as the $\mathsf{LOCAL}+\mathsf{NCC}$ model.

2.2. Preliminaries

We begin this section by introducing some general definitions and concepts from graph theory and then provide the main probabilistic arguments used in this dissertation.

Basic Graph Theory We first review some basic concepts from graph theory. Let $G = (V, E, w)$ be an undirected graph with positive edge weights $w : E \to \mathbb{N}$ (throughout this thesis, $\mathbb{N}$ always refers to the set of positive natural numbers and does not include 0). If $w(e) = 1$ for all $e \in E$, then we call G *unweighted.* To distinguish between the edge sets of different graphs, we will sometimes refer to the nodes and edges of G as $V[G]$ and $E[G]$, respectively. The *neighborhood* of a node $u \in V$ is defined as $N_G(u) := \{v \in V \mid \{u, v\} \in E\}$, and $\deg_G(u) := |N_G(u)|$ denotes its *degree.* The degree of a graph G is defined as $\Delta(G) := \max_{u \in V} \deg_G(u)$, and $\overline{\deg}(G) := \sum_{u \in V} \deg_G(u)/n$ is the average degree of all nodes. We call two nodes $u, v \in V$ *adjacent*, if $\{u, v\} \in E$, and we call two edges adjacent if they share one endpoint. Furthermore, we call an edge $e \in E$ *incident* to a node $v \in V$ if v is one of the endpoints of e.

For any $u, v \in P$, a *path* P between u and v is a sequence of edges

$$P = (\{u = v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{k-1}, v_k = v\}).$$

Throughout this thesis, we will sometimes refer to a path as the set of its edges $P \subseteq E$, or represent P by the sequence of nodes contained in P in the order in which they are *visited* by P, i.e., $P = (u = v_1, v_2, \dots, v_k = v)$. If $u = v$, then P is a *cycle.* We call a path (or cycle) *simple*, if each node in P is visited exactly once (and, in a cycle, $u = v$ is visited only twice).

We say G is *connected* if there is a path P between any two nodes. The *connected components* of G are its maximal connected subgraphs.

We define the *length* of P as $w(P) := \sum_{e \in P} w(e)$. The *distance* between any two nodes $u, v \in V$ is defined as

$$d_G(u, v) := \min_{u\text{-}v\text{-path } P} w(P).$$

A path $P \subseteq E$ between two nodes u and v such that $w(P) = d_G(u, v)$ is called a *shortest path.* The *diameter* of G is defined as the maximum length of any shortest shortest path, i.e.,

$$D(G) := \max_{u,v \in V} d_G(u, v).$$

Alternatively, the diameter can also be defined as the maximum *eccentricity* $\mathrm{ecc}_G(u)$ of any node $u \in V$, which is defined as

$$\mathrm{ecc}_G(u) := \max\{d(u, v) \mid v \in V\}.$$

If P contains k edges, i.e., $|P| = k$, we say P has k *hops.* The *hop-distance* between two nodes u and v is defined as

$$\mathrm{hop}_G(u, v) := \min_{u\text{-}v\text{-path } P} |P|.$$

In addition to the definition of the (weighted) diameter $D(G)$, we define $\mathfrak{D}(G)$ as the *hop-diameter*

$$\mathfrak{D}(G) := \max_{u,v \in V} \operatorname{hop}_G(u,v).$$

The *shortest path hop-distance* between u and v is

$$\operatorname{sph}_G(u,v) := \min_{\text{shortest } u\text{-}v\text{-path } P} |P|,$$

and the *h-limited distance* from u to v is

$$d_{h,G}(u,v) := \min_{\substack{u\text{-}v\text{-path } P \\ |P| \leq h}} w(P).$$

If there is no u-v path P with $|P| \leq h$, then let $d_{h,G}(u,v) := \infty$. Finally, the *shortest-path diameter* $\mathsf{SPD}(G)$ is the minimum number such that $d_{\mathsf{SPD}(G),G}(u,v) = d_G(u,v)$ for all $u, v \in V$. Since all edge weights are at least 1, we have that $D(G) \geq \mathsf{SPD}(G) \geq \mathfrak{D}(G)$. If G is unweighted, then $D(G) = \mathsf{SPD}(G) = \mathfrak{D}(G)$; therefore, in this case we only refer to the diameter $D(G)$ of a graph.

A *tree* T is a connected graph that does not contain any cycle, and a *forest* is a set of trees. It is easy to see that a tree contains $|V[T]| - 1$ edges and a unique path between any two nodes. Any node with degree $\deg_T(v) \leq 1$ is called a *leaf*, otherwise it is called an *inner node*.

We call a tree T *rooted* at some node $s \in V[T]$, if each edge in T is assigned a direction towards the root, i.e., $\{u, v\}$ is directed from u to v if and only if $d_T(s,v) < d_T(s,u)$, in which case we call v the *parent* of u and u a *child* of v. If for $u, w \in V[T]$ there exists a path $P = (u = v_1, v_2, \ldots, v_k = w)$ in T such that $\{v_i, v_{i+1}\}$ is directed from v_i to v_{i+1} for all $1 \leq i < k$, then w is an *ancestor* of v and v is a *descendant* of w. By definition, v is both an ancestor and a descendant of itself. We will refer to the subgraph of T induced by all of v's descendants as the *subtree of* v. The *depth* of v is its distance $d_T(s,v)$ to s, whereas its *height* is defined as the maximum distance from v to any of its descendants D_v, i.e., $\max_{u \in D_v} d(v,u)$. The height of T is the height of its root node.

We parameterize the different graph properties by G whenever we need to differentiate between different graphs. If the graph G is clear from the context, we drop the parameter in the notations above.

Nash-Williams Forest Decomposition Clearly, if the global capacity in our hybrid model is very low, nodes can only communicate with few other nodes at the same time using global edges. For many of our algorithms, the global communication that needs to be performed is prescribed by some given graph; if this graph has a high degree, then carrying out the required communication is hard. Therefore, we often make use of the so-called *Nash-Williams forest decomposition* technique, which allows our algorithms to depend on the *arboricity* of the graph instead of its degree, which is, in many cases, much lower. Before we describe the Nash-Williams forest decomposition from a high level, we establish some additional graph theoretical notions.

Formally, the *arboricity* $a(G)$ of an undirected graph G is the minimum number of forests into which its edges can be partitioned (again, if the graph is clear from the context, we omit the parameter G). Since the edges of any graph with maximum degree Δ can be greedily assigned to Δ forests, $a \leq \Delta$. Furthermore, since the average degree of a forest is at most 2, and the edges of G can be partitioned into a forests, $\overline{\deg} \leq 2a$. Graphs of many important graph families have small arboricity although their maximum degree might be unbounded. For example, a tree obviously has arboricity 1. A cycle can be partitioned into a line and one additional edge, and therefore has arboricity 2. Pseudotrees and cactus graphs, which we consider in Chapter 5, obviously have arboricity 2 as well. Nash-Williams [Nas64] showed that the arboricity of a graph G is given by $\max_{H \subseteq G}(m_H/(n_H - 1))$, where $H \subseteq G$ is a subgraph of G with at least two nodes and n_H and m_H denote the number of nodes and edges of H, respectively. Since any planar graph has at most $3n - 6$ edges by Euler's formula, planar graphs have arboricity at most 3. Moreover, any graph with *genus* g, which is the minimum number of handles that must be added to the plane to embed the graph without any crossings, has arboricity $\mathcal{O}(\sqrt{g})$ [BE10]. Furthermore, it is known that the family of graphs that *exclude a fixed minor* [DL98] and the family of graphs with bounded *treewidth* [DW07] have bounded arboricity.

An *orientation* of G is an assignment of *directions* to each edge, i.e., for every $\{u, v\} \in E$ either $u \to v$ (u is directed to v) or $v \to u$ (v is directed to u). If $u \to v$, then u is an *in-neighbor* of v and v is an *out-neighbor* of u. For $u \in V$ define $\mathrm{N}_{\mathrm{in}}(u) = \{v \in V \mid v \to u\}$ and $\mathrm{N}_{\mathrm{out}}(u) = \{v \in V \mid u \to v\}$. The *indegree* of a node u is defined as $\deg_{\mathrm{in}}(u) = |\mathrm{N}_{\mathrm{in}}(u)|$ and its *outdegree* is $\deg_{\mathrm{out}}(u) = |\mathrm{N}_{\mathrm{out}}(u)|$. A *$k$-orientation* is an orientation with maximum outdegree k.

For a graph with arboricity a, there always exists an a-orientation: We root each tree of every forest arbitrarily and direct every edge from child to parent node. As such a forest is usually not known, the *Nash-Williams forest decomposition technique* helps us to construct a $(2 + \varepsilon)a$-orientation (see, e.g., [BE10]). The technique partitions the nodes of the graph into disjoint sets $V_1, \ldots, V_k$, $k = \mathcal{O}(\log n)$, in the following way. In phase $i \in \{1, \ldots, k\}$, all nodes that have degree at most $(2 + \varepsilon) \cdot a$ are removed from the graph and join the set V_i. The desired orientation is obtained by directing each edge $\{u, v\} \in E$, $u \in V_i$, $v \in V_j$, from u to v if $i < j$, or $i = j$ and $\mathrm{id}(u) < \mathrm{id}(v)$. Throughout this thesis, we will use different implementations of this simple procedure to locally distribute the edges of a graph such that each node is only assigned at most $\mathcal{O}(a)$ edges, which makes our algorithms very efficient in graphs that have low arboricity.

Aggregate Functions Many of our algorithms rely on *aggregate functions*. Formally, an aggregate function f maps a multiset $S = \{x_1, \ldots, x_N\}$ of input values to some value $f(S)$. For some functions f it might be hard to compute $f(S)$ in a distributed fashion, so we will focus on so-called *distributive aggregate functions*: An aggregate function f is called distributive, if there is an aggregate function g (which we call *sub-aggregate function to f*) such that for any multiset S and any partition $S_1, \ldots, S_\ell$ of S, $f(S) = g(f(S_1), \ldots, f(S_\ell))$.

Distributive aggregate functions have the property that they can be computed by combining subresults, which makes it easy to compute them in a distributed fashion. Classical examples of distributive aggregate functions are MAX, MIN, and SUM. For these examples, f and g are the same function. For the COUNT function, as a

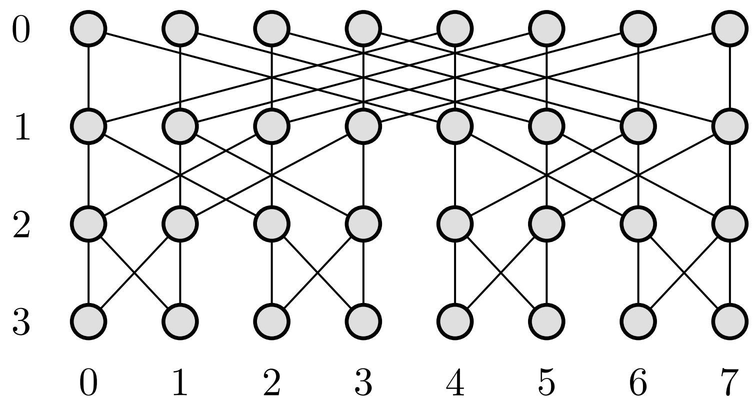


Figure 2.1.: A 3-dimensional butterfly network.

different example, f returns the number of elements in S, whereas g takes the sum of the subresults.

Butterfly Network Some of our algorithms construct *butterfly networks* using the global network. Formally, for $d \in \mathbb{N}$, the *d-dimensional butterfly* is a graph with node set $[d+1] \times [2^d]$, where we denote $[k] = \{0, \ldots, k-1\}$, and an edge set $E_1 \cup E_2$ with

$$\begin{aligned} E_1 = &\{\{(i,\alpha),(i+1,\alpha)\} \mid i \in [d],\ \alpha \in [2^d]\}, \\ E_2 = &\{\{(i,\alpha),(i+1,\beta)\} \mid i \in [d],\ \alpha,\beta \in [2^d], \\ &\quad \alpha \text{ and } \beta \text{ differ only at the } (i+1)\text{-th bit from left to right}\}. \end{aligned}$$

The node set $\{(i,j) \mid j \in [2^d]\}$ represents *level* i of the butterfly, and node set $\{(i,j) \mid i \in [d+1]\}$ represents *column* j of the butterfly. We will refer to level 0 of the butterfly as the *top level*, and to level d as the *bottom level*. An example of a butterfly network is given in Figure 2.1.

Probability Theory We say an event occurs *with high probability* (abbreviated as w.h.p), if it occurs with probability at least $1 - n^{-c}$ for any given constant $c \geq 1$. To show that the correctness and runtime of all randomized algorithms in this thesis hold with high probability, we make use of different concentration inequalities known from literature.

Lemma 2.1 (Markov's Inequality)**.** *Let X be a nonnegative random variable and $k > 0$. We have that*

$$\Pr[X \geq k] \leq \mathbb{E}[X]/k.$$

If X is a sum of sufficiently independent random variables, then we can make use of the following variants of the *Chernoff bounds* (see, e.g., [SSS95]), which give exponentially better bounds than Markov's inequality.

Lemma 2.2 (Chernoff Bound)**.** *Let $X_1, \ldots, X_n$ be k-wise independent random variables with $X_i \in [0,b]$ and let $X = \sum_{i=1}^n X_i$. Then it holds for all $\delta \geq 1$, $\mu \geq E[X]$, and $k \geq \lceil \delta\mu \rceil$*

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

Furthermore, for independent binary random variables X_i, $X = \sum_{i=1}^{n} X_i$, and $0 \leq \delta \leq 1$ and $\mu \leq E[X]$, we have that

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

Furthermore, the well-known *union bound* allows us to show that if a polynomial number of events hold individually w.h.p., then *all* events occur w.h.p., which is formalized in the following lemma.

Lemma 2.3 (Union Bound)**.** *Let $E_1, \ldots, E_k$ be events, each taking place w.h.p. For k polynomial in n, the event $E := \bigcap_{i=1}^{k} E_i$ also takes place w.h.p.*

To apply the union bound to our algorithms and argue that they are correct throughout their entire runtime, w.h.p., they must not run for more than a polynomial number of rounds. To ensure this, we assume that *all* parameters and variables introduced in this thesis, if not stated otherwise, are at most polynomial in n.

Shared Randomness Many algorithms in this thesis require the knowledge of common independent pseudo-random hash functions, which we simply refer to as *random hash functions*. To agree on such functions, for simplicity we assume that the nodes have access to *shared randomness*. However, it can be shown that in fact it suffices to use $\Theta(\log n)$-wise independent hash functions (see, e.g., [Cel+13] and the references therein). Whenever we aim to show that the outcome of a random experiment deviates from the expected value by at most $\mathcal{O}(\log n)$, w.h.p., we can simply apply Lemma 2.2. However, if the deviation we aim to show is higher, we can partition events in a suitable way so that we only need $\Theta(\log n)$-wise independence for each subset of events, and the sum of the deviations does not exceed the overall desired deviation. To agree on such hash functions, all nodes that need to agree on the hash functions have to learn $\Theta(\log^2 n)$ random bits. Although we do not explicitly mention it, whenever common hash functions are used in our algorithms, the required number of bits can easily be broadcast using overlay topologies that have already been established at that point.

2.3. Problem Definitions

This section formally introduces some of the problems that appear repeatedly throughout this thesis. Furthermore, we review some of the classical approaches to solve these problems.

Minimum Spanning Trees First and foremost, many of our algorithms rely on the computation of *spanning trees*. A spanning tree of a connected undirected graph $G = (V, E)$ is a connected subgraph that is a tree and contains all nodes of V. Clearly, a spanning tree contains $n-1$ edges. If G is not connected, then a *spanning forest* contains a spanning tree of each component of G. If G is weighted, then a *minimum spanning tree*, or MST, is a spanning tree that minimizes the sum of all edge weights. Formally, an MST M is a spanning tree of G such that there does not exist a spanning tree M' of G such that $\sum_{e \in E[M']} w(e) < \sum_{e \in E[M]} w(e)$.

Two classical algorithms to compute an MST are *Prim's algorithm* [Pri57] and *Kruskal's algorithm* [Kru56]. In this thesis, we mostly rely on *Borůvka's algorithm* [NMN01], which is one the most famous approaches to the MST problem used in distributed computing. From a high level, the algorithm works as follows. Initially, every node forms a component on its own. In each iteration, every component identifies a *lightest outgoing edge*, which is a minimum-weight edge incident to a node of the component whose other endpoint lies in a different component. The selection of edges induces a forest of components in the graph, where each tree of components forms a new component in the next iteration. Since the number of components halves in each iteration, after $\mathcal{O}(\log n)$ iterations there is only a single component. Furthermore, the set of all selected edges forms an MST of G.

Shortest Path Problems Several chapters of this thesis revolve around *shortest path problems*. The goal of the *All-Pairs Shortest Paths (APSP) Problem* is to let each node $v \in V$ learn $d(v, u)$ for all $u \in V$. In the *Single-Source Shortest Paths (SSSP) Problem*, there is given source node $s \in V$, and each node $v \in V$ has to learn $d(s, v)$. Similarly, in the *h-limited SSSP Problem*, each node $v \in V$ has to learn the h-limited distance $d_h(s, v)$ to v; the SSSP Problem is a special case of this problem for $h = \mathsf{SPD}$. h-limited SSSP can also be extended to multiple sources $S \subseteq V$, where the goal of each node $v \in V$ is to learn $d_h(s, v)$ for all $s \in S$, which we refer to as the *(h, k)-SSP Problem*. In the *Diameter Problem*, every node wants to learn the diameter D of G.

We will also consider approximate solutions of some of the above problems. Formally, we say an algorithm computes an α-approximation of SSSP, if every node $v \in V$ learns an estimate $\widetilde{d}(s, v)$ such that $d(s, v) \leq \widetilde{d}(s, v) \leq \alpha \cdot d(s, v)$. Similarly, for an α-approximation of the diameter, every node $v \in V$ has to compute an estimate $\widetilde{D}$ such that $D \leq \widetilde{D} \leq \alpha \cdot D$.

Two of the most famous approaches to shortest path problems are the *Bellman-Ford algorithm* [For56; Bel58] and *Dijkstra's algorithm* [Dij59]. The former, in particular, admits a natural distributed implementation, upon which many of our algorithms rely: Start with a source node $s \in V$ with $d(s, s) = 0$, and $d(s, v) = \infty$ for all $v \in V \setminus \{s\}$. In each round, every node sends its current distance value to all of its neighbors. Whenever a node v receives $d(s, u)$ from u, it updates $d(s, v)$ to $\min\{d(s, v), d(s, u) + w(\{u, v\})\}$. After SPD rounds, every node knows its distance to s.

Shortest-Path Trees To efficiently solve shortest path problems, many of our algorithms rely on the computation of *shortest-path trees*. A *breadth-first search (BFS) tree* [Pel00] is a tree that is rooted at a given source node $s \in V$ and that contains for each node $v \in V$ a unique path P from s to v such that $|P| = hop(s, v)$. We require every node $v \in V$ to know its parent $\pi(v)$ in the BFS tree. The weighted counterpart of a BFS tree is called *shortest-path tree*. Such a tree contains, for a given source $s \in V$ and any node $v \in V$, a unique path P from s to v such that $w(P) = d(s, v)$, and each node $v \in V$ knows its parent $\pi(v)$ in the tree.

A BFS tree can easily be computed using a distributed breadth-first search. In the first iteration, the source node s sends a token to all of its neighbors. Whenever a node $v \in V$ receives a token for the first time from some neighbor $u \in N(v)$, it sets u as its parent $\pi(v)$. Whereas this obviously constructs a BFS tree in $\mathcal{O}(\mathfrak{D})$ many rounds, computing a shortest-path tree is more complex. Specifically, tokens may reach nodes first over paths that contain few hops, but actually have a large distance. This implies that any naive approach, such as using a variant of the distributed Bellman-Ford algorithm, requires SPD many rounds. Furthermore, this also means that, in contrast to a BFS, nodes may have to send tokens *multiple* times if they learn a shorter path.

3 Fast Construction of Overlay Networks

To begin our study of hybrid networks, we consider the most restricted situation in which each node *only* knows its neighbors in the local network G. Arguably, such a network is of limited utility, as the nodes can only rely on their local connections at the beginning. To solve meaningful tasks such as computing aggregates, monitoring network properties [Gmy+17a], or quickly reconfiguring the network to cope with churn or adversarial behavior [DGS16; AS18], the global network needs to form some suitable topology. Our goal is to transform the initial network into such a topology using the global edges and the fact that the global network behaves like an *overlay network*. That is, we can establish *new* global edges by sending around node identifiers and, in principle, obtain any desired topology.

In this chapter, we consider the $\mathsf{CONGEST}+\mathsf{NCC}_0$ model, which allows local capacity $\lambda = \mathcal{O}(1)$ and global capacity $\gamma = \mathcal{O}(\log n)$. Under this limited communication, constructing a suitable overlay becomes an intricate problem. Specifically, we cannot simply introduce nodes to one another until the network form a clique, which would require each node to send $\Theta(n)$ messages over global edges. The problem of constructing a suitable overlay network under limited communication at each node is also known as the *Overlay Construction Problem*. In this chapter, we specifically aim at constructing a *well-formed tree*, which is an unweighted rooted tree that has constant degree and height $\mathcal{O}(\log n)$. Such a tree can easily be rearranged into many other classical network topologies such as butterfly networks, path graphs, sorted rings, or De Bruijn Graphs, which makes it a suitable choice for our desired topology.

The Overlay Construction Problem has already received some attention in the literature. First of all, there is a fundamental lower bound of $\Omega(\log n)$ time steps to construct *any* overlay of logarithmic diameter even if the nodes were allowed to perform an arbitrary amount of communication. To illustrate this, consider an overlay that forms a linked list. Even if all nodes exhaustively exchange their neighborhoods in every round as proposed above, this can only halve the diameter in every iteration. Hence, it takes $\Omega(\log n)$ rounds until the diameter is logarithmic. To the best of our knowledge, the first algorithm that can be used to solve the Overlay Construction Problem in our model is an $\mathcal{O}(\log^2 n)$ time algorithm by Angluin et al. [Ang+05]. Later, Aspnes and Wu applied the approach to graphs with outdegree 1 and achieve a runtime of $\mathcal{O}(\log n)$, w.h.p. [AW07]. More recently, we presented a deterministic $\mathcal{O}(\log^2 n)$-time solution in a model that allows polylogarithmic communication [Gmy+17a], which, as we point out in Chapter 5, can also be used to compute a *minimum spanning tree* (MST) of the initial graph. With little effort, the algorithm can be adapted to our hybrid model with the same runtime. Even more recently, we presented an algorithm for the Overlay Construction Problem that takes optimal time $\mathcal{O}(\log n)$, w.h.p., but requires polylogarithmic global communication [Göt+20]. In stark contrast to the previous approaches, the algorithm heavily relies on rapid random walks to establish random connections until the graph becomes an expander.

In this chapter, we present an algorithm that, starting from an arbitrary connected graph in the $\mathsf{CONGEST}+\mathsf{NCC}_0$ model, constructs a well-formed tree of global edges in time $\mathcal{O}(\log^{3/2} n)$, w.h.p. Inspired by the classical approach of grouping and merging large clusters of nodes (see [Ang+05; AW07; Gmy+17a]), our algorithm always identifies *large* sets of clusters to merge, which reduces the overall number of phases required until a single cluster remains. We internally organize each cluster as a low-diameter butterfly network, which allows us to perform the grouping and merging operations very efficiently, and from which the final well-formed tree can easily be derived. Furthermore, we derive a spanning tree of G from the outcome of our algorithm.

Underlying Publication The chapter is based on the following publication.

> T. Götte, K. Hinnenthal, and C. Scheideler. "Faster Construction of Overlay Networks". In: *International Colloquium on Structural Information and Communication Complexity (SIROCCO).* 2019, pp. 262–276 [GHS19]

Note that in the above publication, our algorithm is presented in an overlay network model in which each node can communicate polylogarithmically many bits in total, including to its neighbors in G. In this chapter, we present all additional details necessary to translate the algorithm to the $\mathsf{CONGEST}+\mathsf{NCC}_0$ model. Furthermore, the chapter describes a more complex grouping and merging phase that facilitates the construction of a spanning tree, which cannot be done with the original algorithm.

We remark that the results of this chapter were published prior to our recent $\mathcal{O}(\log n)$-time algorithm. However, the algorithm presented in this chapter is more efficient regarding its communication complexity, as it requires global capacity $\mathcal{O}(\log n)$ instead of $\mathcal{O}(\log^3 n)$. Furthermore, whereas our more recent algorithm inherently uses randomization, the algorithm in this chapter may admit a deterministic implementation.

Outline We begin the technical part of this chapter by introducing two algorithmic primitives that we heavily use in this chapter, and upon which multiple algorithms in the upcoming chapters rely. In Section 3.2, we then describe a preliminary step to our main algorithm. More precisely, we give a distributed algorithm that transforms the initial graph G into a sparse graph of degree $\mathcal{O}(\log n)$ based on an efficient *spanner construction* [EN18]. We then describe our main algorithm, which is comprised of a sequence of *subphases*, from a high level in Section 3.3. The main part of this chapter is the technical description of a single subphase, which we provide in Section 3.4. The outcome of this phase will be a spanning tree of our constructed sparse graph, which may contain global edges that are not actually in G. To obtain a well-formed tree from this spanning tree, as well as a spanning tree of our actual graph G, we give a simple protocol in Section 3.5 that applies some additional observations on our main algorithm. We conclude the chapter with an outlook in Section 3.6.

Related Work The problem studied in this chapter relates to research in overlay networks in general, which started in the early 2000s. Some popular examples for these early overlays are Chord [Sto+01], Pastry [RD01], and skip graphs [AS07]. However, most research on overlay networks focuses on the problem of efficiently joining and leaving such an overlay, or keeping it in some legal state despite some potentially heavy churn (e.g., [Aug+15; AS18; GVS19]). Still, adversarial nodes and churn beyond the limits prescribed in these papers may push an overlay into a corrupted state from which a valid topology may need to be recovered. If we are guaranteed that the initial state of the nodes is *not* corrupted, algorithms for network constructions like ours may be used [Ang+05; AW07; Gmy+17a]. If that is not the case, however, then solutions for *self-stabilizing overlay networks* may be applicable.

There is a rich collection of papers on self-stabilizing overlays (see [FSS20] for a comprehensive overview). Since it is a difficult problem to recover from *arbitrary* initial states, even more so in the asynchronous message passing model that is often considered for self-stabilization, most existing results do not provide time or work bounds but instead focus on proving that convergence is possible at all. If convergence times are proven, however, these are often much higher than polylogarithmic, such as the $\mathcal{O}(n)$ time bound for self-stabilizing lists [ORS07] or the $\mathcal{O}(n^3)$ time bound for self-stabilizing Delaunay graphs [Jac+12]. Notable exceptions are [Jac+14; BGP13]: In [Jac+14], the authors show a convergence time of $\mathcal{O}(\log^2 n)$ rounds for the SKIP+ graph, and in [BGP13] the authors present a general framework for the self-stabilizing construction of a large class of overlays that can be used, for example, to achieve a convergence time of just $\mathcal{O}(\log n)$ for SKIP+ graphs. However, no low bounds for the communication work are known; in fact, the work required for the constructions in [Jac+14; BGP13] can be prohibitively large.

3.1. Algorithmic Primitives

The algorithm in this chapter relies on two primitives, which will be used in different forms throughout the entire thesis. For the primitives, we assume that the nodes simulate a d-dimensional butterfly in the global network. How the butterfly is precisely constructed and simulated will depend on the specific algorithm; therefore, we present our primitives in a very general form. More precisely, we assume that the butterfly nodes can send and receive messages to and from their at most four neighbors in synchronous rounds. For our primitives, we give a bound on the number of messages each node is required to send in each round, which will allow us to efficiently simulate the algorithms in the global network.

Aggregate-and-Broadcast Algorithm In an *Aggregate-and-Broadcast Problem*, we are given a multiset A of *input values* to a distributive aggregate function f with sub-aggregate function g, and every node of the butterfly's top level (level 0) stores exactly one input value. The goal is to let every butterfly node learn $f(A)$.

We solve the problem using the following simple *Aggregate-and-Broadcast Algorithm*. First, every node of the top level that stores an input value x creates a packet that contains the value $g(x)$. Note that there exists one unique path between any node of the butterfly's top level and any node of its bottom level. Therefore, we can easily route each packet from the top level to the first node of the bottom level (i.e., the node $(d, 0)$) along the respective path. If two packets with values x and y reach the same node of level i at the beginning of round $i+1$, only one packet that contains the value $g(x, y)$ is forwarded to level $i + 1$. After d rounds, node $(d, 0)$ knows $f(A)$. Finally, we broadcast this value from $(d, 0)$ for $2d$ rounds in the butterfly, whereby every butterfly node learns $f(A)$.

Theorem 3.1 (Aggregate-and-Broadcast)**.** *The Aggregate-and-Broadcast Algorithm solves any Aggregate-and-Broadcast Problem within* $3d$ *rounds. In each round, every butterfly node sends and receives at most* 4 *messages.*

Route-and-Combine Algorithm In a *Route-and-Combine Problem*, we are given a multiset of input values A to a distributive aggregate function f with sub-aggregate function g. The input values are partitioned into k *routing groups* $R_1, \ldots, R_k \subseteq A$. Each input value $x \in A$ is stored by some node s_x of the butterfly's top level such that every node only stores $\mathcal{O}(\log n)$ many input values of *different* routing groups, i.e., $|\{x \in A \mid s_x = v\}| = \mathcal{O}(\log n)$ for every butterfly node v. Every routing group R_i has a target t_i, which is a node of the bottom level of the butterfly, such that each node is only target of at most $\mathcal{O}(\log n)$ routing groups. Formally, $|\{i \in \{1, \ldots, k\} \mid t_i = v\}| = \mathcal{O}(\log n)$ for every butterfly node v of the bottom level. The goal is to let each target t_i learn $f(R_i)$.

To solve any Route-and-Combine Problem, we use the following *Route-and-Combine Algorithm.* First, every node u of the butterfly's top level combines all values of the same routing group R_i stored at it and creates a single packet $p_{u,i}$ using g. We then aggregate all packets $p_{u,i}$ corresponding to the same routing group R_i at some randomly chosen *intermediate target* t'_i of the butterfly's bottom level, which we choose by using a (pseudo-)random hash function $h : \{1, \ldots, k\} \to [2^d]$. For the aggregation, we simply route each packet $p_{u,i}$ to its intermediate target t'_i along the unique path towards t'_i. Whenever two packets that correspond to the same routing group (and, consequently, have the same intermediate target) arrive at the same node, they are combined into a single packet. After d rounds, every t'_i receives $f(R_i)$.

To route $f(R_i)$ to t_i, we first move it from t'_i to the topmost node of the same column in d rounds. From there, it is routed along the unique path to t_i in an additional d rounds.

Theorem 3.2 (Route-and-Combine)**.** *The Route-and-Combine Algorithm solves any Route-and-Combine Problem within* $3d$ *rounds. In each round, at most one node of each column sends a message, and all other nodes of the column do not send any messages. Furthermore, every node sends at most* $\mathcal{O}(\log n)$ *many messages in each round, w.h.p.*

Proof. It is easy to see that the algorithm solves any Route-and-Combine Problem. If each node only sends $\mathcal{O}(\log n)$ messages in each round, packets can always be sent up and down in the butterfly without ever being delayed, wherefore in each round solely the nodes of some level i send and receive packets.

It remains to show that each node only sends $\mathcal{O}(\log n)$ messages. Since initially each node of the butterfly's top level stores at most $\mathcal{O}(\log n)$ messages of different routing groups, after creating packets $p_{u,i}$ every node stores at most $c \log n$ many packets for some constant $c \in \mathbb{N}$. Consider some fixed edge e between level i and level $i+1$ of the butterfly. Let S be the set of nodes u of level 0 such that the unique path from u to any node of the bottom level contains e. Since the nodes of each level $i' > 0$ have exactly two neighbors in level $i'-1$, S contains 2^i nodes. By the same argument, there are exactly $2^{d-(i+1)}$ many nodes T of the bottom level reachable via e when only moving downwards. Note that only the paths from S to T contain the edge e. Let $R_1, \dots, R_l$ be the routing groups for which at least one packet of each R_j is located at a node of S, and X_j be the binary random variable that indicates whether the intermediate target of R_j lies in T. Since the intermediate target of each routing group is chosen uniformly at random, $\Pr[X_j = 1] = 2^{d-(i+1)}/2^d = 1/2^{i+1}$. Furthermore, since the number of different routing groups in S is $l \leq 2^i \cdot c \log n$, $X = \sum_{j=1}^{l} X_j$ is a sum of binary independent random variables with expected value $\mathbb{E}[X] \leq 2^i \cdot c \log n / 2^{i+1} \leq c \log n$. By the Chernoff Bound in Lemma 2.2, we have

$$\Pr[X \geq (1+\delta) c \log n] \leq e^{-\delta c \log n/3} \leq n^{-\delta c/3} \leq n^{-c'}$$

for constant $\delta \geq 1$ such that $\delta \geq 3c'/c$. Therefore, $X = \mathcal{O}(\log n)$, w.h.p. Since packets that correspond to the same routing group are combined, only $\mathcal{O}(\log n)$ packets will be sent over e. By taking the union bound over all edges, we obtain that every node will only send $\mathcal{O}(\log n)$ messages in each round when routing packets to their intermediate targets. Since this also implies that every intermediate target only stores $\mathcal{O}(\log n)$ packets, every node only has to send $\mathcal{O}(\log n)$ messages when sending the results upwards within each column, w.h.p.

Finally, when routing the packet that contains $f(R_j)$ to t_j, we can apply the Chernoff argument from above in a very similar way. We fix an edge e, and define S and T as above. Let $R_1, \dots, R_l$ be the routing groups such that t_j lies in T for all $j \in \{1, \dots l\}$, and let X_j be the binary random variable indicating whether $f(R_j)$ is stored by some node in S. Since the intermediate targets have been chosen uniformly at random, we have that $\Pr[X_j = 1] = 2^i/2^d$ for every $j \in \{1, \dots, l\}$. By assumption, there exists a constant c such that t_j is target of at most $c \log n$ routing groups, i.e., $l \leq 2^{d-(i+1)} \cdot c \log n$. Therefore, $X = \sum_{j=1}^{l} X_j$ is a sum of binary independent random variables with expected value $\mathbb{E}[X] = 2^{d-(i+1)} \cdot c \log n \cdot 2^i/2^d \leq c \log n$. The Chernoff bound can be applied in the same way as above to argue that at most $\mathcal{O}(\log n)$ packets will be sent over e. Taking the union bound over all edges yields the theorem. □

3.2. Transforming G into a Sparse Graph

As a preliminary step to our main algorithm, G needs to be transformed into a graph that has degree $\mathcal{O}(\log n)$ in the global network. Our approach can be divided into two steps. In Step 1, we compute an $\mathcal{O}(\log n)$-*spanner* $G_1 = (V, E_1)$ of G that has $\mathcal{O}(n \log n)$ edges using the algorithm of Elkin and Neiman [EN18]. Formally, a subgraph $G' = (V, E')$, $E' \subseteq E$, is called a (multiplicative) α-spanner of G if

$d_{G'}(u,v) \le \alpha \cdot d_G(u,v)$ for all $u,v \in V$ (we say G' has *stretch* α). As a byproduct, the algorithm yields an $\mathcal{O}(\log n)$-orientation in G_1. Using this orientation, in Step 2 we redirect the edges of G_1 to obtain a graph $G_2 = (V, E_2)$ in the global network that has degree $\mathcal{O}(\log n)$.

Step 1: Construct a Spanner G_1 For our purpose, the algorithm of Elkin and Neiman [EN18] can be stated as follows. First, every node $u \in V$ samples a value r_u that is exponentially distributed with parameter $\beta = 1/2$ and sends it to its neighbors in G. Whenever at the beginning of any subsequent round a node v receives a message that contains the value r_u, it stores $m_u(v) = r_u - d(v,u)$. Note that since G is unweighted, $d(v,u) = \text{hop}(v,u)$ can be inferred from the number of rounds it takes for r_u to reach v. Let $p_u(v)$ denote the neighbor of v from which it first received r_u, breaking ties arbitrarily. To conclude this round, v sends the current value of $m(v) = \max_{u \in V} m_u(v)$ to all of its neighbors in G. After $k = \mathcal{O}(\log n)$ rounds, each node v adds to the spanner G_1 the set of edges

$$C(v) = \{\{v, p_u(v)\} \mid m_u(v) \ge m(v) - 1\}.$$

To obtain an orientation of G_1, each node v directs away all the edges in $C(v)$. To break ties, any edge $\{u,v\}$ that is contained in both $C(u)$ and $C(v)$ becomes directed towards the endpoint that has higher identifier.

Lemma 3.3. *The algorithm constructs an $\mathcal{O}(\log n)$-spanner $G_1 = (V, E_1)$ of G such that $|E_1| = \mathcal{O}(n \log n)$ in time $\mathcal{O}(\log n)$, w.h.p. Furthermore, it computes an $\mathcal{O}(\log n)$-orientation of G_1, w.h.p.*

Proof. Let $k := 2c \ln n = \mathcal{O}(\log n)$ for some $c > 0$ to be determined. To show that G_1 is an $\mathcal{O}(\log n)$-spanner, we prove that the statement of Claim 3 of the analysis of Elkin and Neiman [EN18] holds, w.h.p. Claim 3 states that $r_u < k$ for all $u \in V$. By the definition of the exponential distribution and by our choice of β and k we have that

$$\Pr[r_u \ge k] = e^{-\beta k} = e^{-2c \ln n/2} = n^{-c},$$

which shows that the statement holds w.h.p. Since the subsequent lemmas of the analysis up to Lemma 6 are implied by this claim, we conclude that the stretch of G_1 is $2k - 1 = \mathcal{O}(\log n)$; in particular, G_1 is connected.

Finally, we argue that $|C(u)| = \mathcal{O}(\log n)$ for all $u \in V$, w.h.p. As described in the proof of Lemma 2 of Elkin and Neiman's analysis, Lemma 1 of their analysis implies that the event $|C(u)| \le t$ happens with probability at most $(1/p)^{t-1}$, where $p = (1 - e^{-\beta})^{-1} \approx 2.54 > 1$. Therefore, by choosing $t = c' \log_p n + 1 = \mathcal{O}(\log n)$ for some $c' > 0$, we have that $|C(u)| = \mathcal{O}(\log n)$, w.h.p. Together with the union bound, we conclude that $|E_1| = \mathcal{O}(n \log n)$ and that we obtain an $\mathcal{O}(\log n)$-orientation, w.h.p. $\square$

Step 2: Construct a Low-Degree Graph G_2 To infer a graph G_2 with degree $\mathcal{O}(\log n)$ from G_1, we let each node $v \in V$ arrange all of its in-neighbors in the computed orientation, of which there can be arbitrarily many, as a list. Additionally, v keeps one of its incoming edges. This idea is conceptually similar to the construction of *child-sibling trees* [Gmy+17a; AW07]. Formally, let $v_1, \dots, v_{k(v)}$ denote the $k(v)$

many in-neighbors of node $v \in V$ sorted by increasing identifier. The edge set of G_2 is defined as

$$E_2 = \{\{v, v_1\} \mid v \in V,\, k(v) \geq 1\} \cup \{\{v_i, v_{i+1}\} \mid v \in V,\, 1 \leq i < k(v)\}.$$

Note that every node has at most $\mathcal{O}(\log n)$ out-neighbors in G_1, w.h.p., by Lemma 3.3, and each node will be assigned at most two neighbors by each of its out-neighbors. Therefore, the degree of G_2 is at most $\mathcal{O}(\log n)$. Clearly, this step can be performed in time $\mathcal{O}(1)$, and we conclude the following lemma.

Lemma 3.4. *$G_2 = (V, E_2)$ is a connected graph with degree $\mathcal{O}(\log n)$, w.h.p., and is computed in time $\mathcal{O}(\log n)$.*

3.3. High-Level Algorithm

Our algorithm to solve the Overlay Construction Problem is divided into consecutive *phases*, where each phase relies on a set of invariants maintained after the previous phase. We first present the algorithm from a high level, and then give the details of a single phase in Section 3.4.

We organize sets of nodes into *supernodes*, where initially each node makes up a supernode on its own. Then, we repeatedly merge supernodes into larger supernodes until only a single supernode containing all nodes of V remains. For each supernode v, we maintain a subtree of G_2 that contains exactly the nodes of v, and which we refer to as the *spanning tree* of v. Multiple supernodes merge by selecting *merge edges* from E_2 that connect their corresponding spanning trees. At the end of our algorithm, the set of all selected merge edges will form a spanning tree of G_2. Furthermore, we maintain an internal $\lfloor \log |v| \rfloor$-dimensional butterfly network in each supernode v whose size $|v|$ is not too large (where v's size $|v|$ is the number of nodes it contains). The internal butterfly will help us to coordinate communication between supernodes. As we show in Section 3.4.3, a spanning tree of size k and degree $\mathcal{O}(\log n)$ can be rearranged into an $\mathcal{O}(\log k)$-dimensional butterfly network in time $\mathcal{O}(\log k)$. Therefore, whenever multiple supernodes merge to form a larger supernode u, we can construct a new butterfly in time $\mathcal{O}(\log |u|)$. Most importantly, we always merge *large* sets of supernodes in a highly coordinated fashion, which, compared to previous approaches, results in a faster growth of supernodes and fewer rounds until only a single supernode remains. Once a single supernode remains, the internal spanning tree can be used to construct a well-formed tree as well as a spanning tree of G, which we show in Section 3.5.

More precisely, our algorithm proceeds in phases $0, \ldots, \lceil \log \log n \rceil - 1$, where the goal of phase i is to grow every supernode of size $|v| \in [2^{2^i}, 2^{2^{i+1}} - 1]$ to a supernode of size at least $2^{2^{i+1}}$. To optimally balance the number of phases with the required runtime of each phase, we further divide each phase i into subphases $0, \ldots, \lceil \sqrt{2^i} \rceil - 1$. Correspondingly, the goal of subphase j of phase i is to grow every *active* supernode, which is a supernode whose size lies in the interval $I = [2^{2^i + j\lceil \sqrt{2^i} \rceil}, 2^{2^i + (j+1)\lceil \sqrt{2^i} \rceil} - 1]$, to a supernode of size at least $2^{2^i + (j+1)\lceil \sqrt{2^i} \rceil}$. If a supernode is of size at least $2^{2^i + (j+1)\lceil \sqrt{2^i} \rceil}$ already at the beginning of subphase j, we call it *inactive*. As we will later show, our algorithm ensures that there are no supernodes of smaller size than $2^{2^i + j\lceil\sqrt{2^i}\rceil}$ at the beginning of subphase j. Therefore, every supernode is either active or inactive in each phase.

Technically, this requires the initial supernodes to be of size 2 already. To ensure this, we can for example let each node $v \in V$ simulate an additional node u whose only neighbor is v, and form a supernode with that simulated node. Our algorithm can easily be performed in the resulting graph with constant simulation overhead.

Lemma 3.5. *If in subphase j of phase i every active supernode grows to a supernode of size at least $2^{2^i + (j+1)\lceil\sqrt{2^i}\rceil}$ in time $c \cdot 2^i$ for some $c > 0$, then the algorithm merges all nodes into a single supernode in time $\mathcal{O}(\log^{3/2} n)$.*

Proof. We only have to prove the overall runtime. Note that phase i consists of $\lceil 2^{i/2}\rceil$ subphases. Therefore, summing up over all phases from 0 to $T = \lceil\log\log n\rceil - 1$ results in

$$\sum_{i=0}^{T} \lceil 2^{i/2}\rceil \cdot c \cdot 2^i \leq c \cdot \lceil\sqrt{\log n}\rceil \cdot \sum_{i=0}^{T} 2^i \leq c \cdot \lceil\sqrt{\log n}\rceil \cdot \log n = \mathcal{O}(\log^{3/2} n).$$

□

3.4. A Single Subphase

We now describe the details of a single subphase of the algorithm. More specifically, we consider subphase j of phase i. The subphase is divided into two stages: In the *Expansion Stage*, the goal of each active supernode is to get to know at least $2^{\lceil\sqrt{2^i}\rceil}$ other active supernodes. In the *Merging Stage*, we use this information to select a set of merge edges between the supernodes that results in larger supernodes. As we will show, each resulting supernode u consists of at least $2^{2^i + (j+1)\lceil\sqrt{2^i}\rceil}$ nodes. Furthermore, the spanning trees of the supernodes that merge into u, together with the respective merge edges, form a spanning tree of u. If u is not too large, we can successfully construct a low-diameter butterfly within u, which allows u to act actively in the next subphase. Otherwise, the stage will prematurely terminate and leave u as an inactive supernode.

3.4.1. Beginning of the Subphase

Before we present the two stages of the subphase in detail, we first formally describe the invariants that hold at the beginning of each subphase. It is easy to see that the invariants hold at the beginning of the first subphase. In the remainder of this section, we show how the invariants can be established for the next subphase, which inductively proves the correctness of our algorithm. In the following, we refer to the nodes contained in a supernode as its *members*. Our first formal proposition is the following.

Proposition 3.6 (Size Lower Bound)**.** *At the beginning of subphase j of phase i, every supernode is of size at least $\min\{2^{2^i + j\lceil\sqrt{2^i}\rceil}, n\}$. Furthermore, the members of each supernode v know whether v is active or inactive.*

Internal Spanning Trees Throughout the algorithm's execution, we maintain a set of edges $\mathcal{E} \subseteq E_2$ that is initially set to $\mathcal{E} = \emptyset$. In each subphase, the set is extended by the merge edges selected between supernodes.

Proposition 3.7 (Internal Spanning Tree)**.** *At the beginning of each subphase, the edges of $\mathcal{E} \subseteq E_2$ form a forest in G_2. For each supernode v (active or inactive), there is exactly one connected component $\mathcal{E}_v$ in $\mathcal{E}$ that contains each member of v and no other node.*

We will refer to the component $\mathcal{E}_v$ as the *internal spanning tree* of v.

Internal Butterflies To facilitate communication between active supernodes, every active supernode is internally organized as a butterfly network. Let v be an active supernode at the beginning of a subphase, i.e., $|v| \leq 2^{2^i+(j+1)\lceil\sqrt{2^i}\rceil} - 1$. The *internal butterfly* B_v of v is a $\lfloor \log |v| \rfloor$-dimensional butterfly network consisting of virtual nodes. Note that since $|v| \leq 2^{2^{i+1}}$, the dimension of B_v is in $\mathcal{O}(2^i)$. The butterfly is simulated by the nodes in the following way. Let $x_0, \ldots, x_{|v|-1} = \ell_v$ be the members of v in some arbitrary order. For $0 \leq l \leq 2^{\lfloor \log |v| \rfloor} - 1$, x_l simulates the nodes of column l of the butterfly (in this case we say x_l is a *node of* B_v). To do so, x_l has to know the identifier of every other member of v that simulates a butterfly node that is adjacent to a butterfly node of column l. If $|v|$ is not a power of 2, then for every $2^{\lfloor \log |v| \rfloor} \leq l \leq |v| - 1$, the node x_l, which does not simulate a column of the butterfly, is connected to the butterfly via a bidirected edge to its *helper node* $x_{l-2^{\lfloor \log |v| \rfloor}}$. An example of an internal butterfly can be found in Figure 3.1.

Let ℓ_v be v's member of highest identifier, and define v's *identifier* $\mathrm{id}(v) := \mathrm{id}(\ell_v)$. For every active supernode, we ensure the following proposition at the beginning of each subphase.

Proposition 3.8 (Internal Butterfly)**.** *Let v be an active supernode at the beginning of a subphase. v's members know id(v) and v's size $|v|$. The members are enumerated from 0 to $|v|-1$, and each member x_l knows its index l, as well as the identifier and index of each member $x_{l'}$ for $l' = l \pm 2^t$ for all $t \in \mathbb{N}_0$, if that node exists.*

Note that this information enables the members of an active supernode v to simulate the internal butterfly B_v as described above.

The members of an *inactive* supernode u, on the other hand, do not simulate an internal butterfly, since they may not possess the necessary information. In fact, they may not even know $\mathrm{id}(u)$ or $|u|$. The only thing we guarantee for an inactive supernode is that its members know that the supernode is inactive.

3.4.2. Expansion Stage

In the Expansion Stage, every active supernode v maintains sets of *discovered nodes* $\Gamma(v)$. We say two supernodes $v \neq u$ are *adjacent in* G_2, if there exist members x and y of v and u, respectively, such that $\{x, y\} \in E_2$. Further, we define the *adjacency graph of the supernodes* $H = (V_H, E_H)$, where V_H is the set of active supernodes at the beginning of the Expansion Stage, and E_H contains an edge $\{u, v\}$ if and only if u and v are adjacent in G_2. At the beginning of the Expansion Stage, $\Gamma(v)$ contains v and all of v's neighbors in H. Therefore, the following proposition holds.

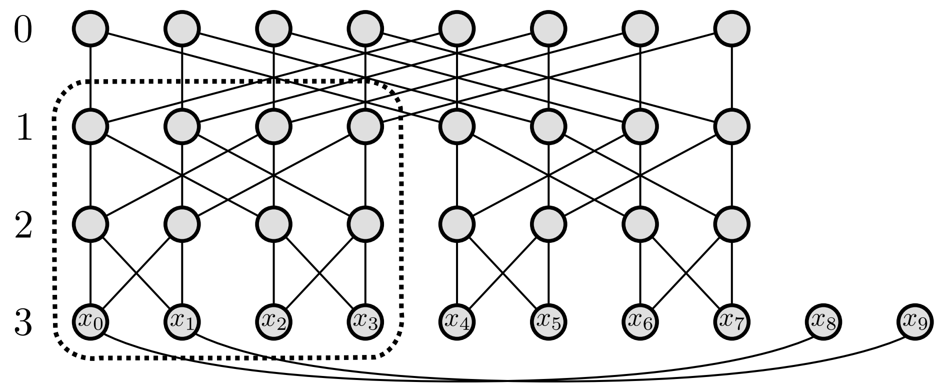


Figure 3.1.: Internal butterfly B_v of a supernode v with $|v| = 10$. The nodes $x_0, \ldots, x_9$ at the bottom are members of v. The first eight nodes $x_0, \ldots, x_7$ construct the 3-dimensional butterfly by simulating one column each. The other two nodes x_8 and x_9 connect to their corresponding helper nodes. Indicated by the dashed outline, the first four nodes $x_0, \ldots, x_3$ simulate the small butterfly b_v, which is described in Section 3.4.2.

Proposition 3.9 (Symmetry of Γ)**.** *$u \in \Gamma(v)$ if and only if $v \in \Gamma(u)$ for all active supernodes u, v.*

We will maintain the proposition throughout the entire Expansion Stage. v's goal is to expand $\Gamma(v)$ until it contains $2^{\lceil \sqrt{2^i} \rceil}$ many active supernodes. This will allow us to find large clusters of supernodes to be merged in the Merging Stage. To do so, v performs $\lceil \sqrt{2^i} \rceil$ *introduction steps*, in each of which it introduces all of its discovered neighbors to one another until $\Gamma(v)$ is large enough.

To carry out the required communication, v utilizes a subgraph of its internal butterfly which we refer to as v's *small butterfly* b_v (see dashed outline in Figure 3.1). More precisely, b_v is the $(2\lceil \sqrt{2^i} \rceil)$-dimensional butterfly that results from taking only the leftmost $2^{2\lceil \sqrt{2^i} \rceil}$ columns and bottommost $2\lceil \sqrt{2^i} \rceil + 1$ rows of B_v. Note that b_v must exist as $|v| \geq 2^{2^i + j\lceil \sqrt{2^i} \rceil} \geq 2^{2\lceil \sqrt{2^i} \rceil} \geq 16$ for $i \geq 2$ (for $i = 1$ we simply choose $b_v = B_v$). We will distributedly store $\Gamma(v)$ within b_v, which allows us to perform each introduction step efficiently. To guarantee that no node of b_v ever has to store too much information, we ensure that $\Gamma(v)$ always contains at most $2^{2\lceil \sqrt{2^i} \rceil}$ many supernodes.

Begin of the Expansion Stage Whereas $\Gamma(v)$ is defined as a set of *supernodes*, internally each supernode v actually stores *representatives* of each supernode $u \in \Gamma(v)$, which are members of u. The representatives are the nodes that actually carry out the communication between supernodes. We will maintain the following proposition at all times.

Proposition 3.10 (Degree of Representatives)**.** *Every node $v \in V$ stores at most $\mathcal{O}(\log n)$ representatives of supernodes. Furthermore, every node is stored as a representative by at most $\mathcal{O}(\log n)$ many supernodes.*

In preparation for the first introduction step, v has to retrieve one representative of each supernode in $\Gamma(v)$, and determine whether $|\Gamma(v)| \geq 2^{\lceil \sqrt{2^i} \rceil}$, in which case v has already achieved the goal of the Expansion Stage. To do so, every node $y \in V$ that is a member of an active supernode u sends $\mathrm{id}(u)$ to all of its neighbors in G_2. Note that this can be done in the global network since G_2 has degree $\mathcal{O}(\log n)$. Thereby, every member x of supernode v learns which of its neighbors lie in different active supernodes, and the identifier of the respective supernode. Every neighbor of x that is contained in a different active supernode is stored as a representative by x. If x is not a node of B_v, x sends its collected representatives to its helper node.

The remaining preliminary steps of v can be divided into three tasks. First, v *filters* the representatives and retains only *one* representative for each supernode in $\Gamma(v)$, then *moves* the remaining representatives into b_v, and finally *replaces* each representative of a supernode u by a a random node of b_u. The first task can be achieved by performing the Route-and-Combine Algorithm in B_v in the following way. Every member x of v that is a node of B_v and that stores some representative y of a supernode u (either because y is a neighbor of x in G_2, or because y has been sent to x) contributes $\mathrm{id}(y)$ as an input value contained in routing group $R_{\mathrm{id}(u)}$, and places its input values at the topmost node of its column in B_v. Therefore, each routing group contains the identifiers of members of the same supernode. By choosing the maximum function MAX as the aggregate function f and sub-aggregate function g, $f(R_{\mathrm{id}(u)})$ will yield a node that has highest identifier of all members of u stored by v; this node remains as the unique representative of u. The target of $R_{\mathrm{id}(u)}$ is chosen uniformly at random among the butterfly nodes of the bottom level using a (pseudo-)random hash function $h : \{0,1\}^* \to [2^{\lfloor \log |v| \rfloor}]$ that is known to all nodes (recall that $[k]$ denotes the set $\{0, \ldots, k-1\}$). An illustration of this step can be found in Figure 3.2a.

Lemma 3.11 (Filtering Representatives)**.** *Filtering the representatives takes time $\mathcal{O}(2^i)$ and maintains Proposition 3.10, w.h.p.*

Proof. Since the degree of G_2 is $\mathcal{O}(\log n)$, Proposition 3.10 clearly holds at the beginning, and each node contributes at most $\mathcal{O}(\log n)$ input values to the Route-and-Combine Problem. Furthermore, since the target of each routing group is chosen uniformly and independently at random, it follows from a simple application of the Chernoff bound of Lemma 2.2 that each node of the bottom level of B_v is target of at most $\mathcal{O}(\log n)$ routing groups, w.h.p., which establishes the preconditions of Theorem 3.2 and shows that Proposition 3.10 is maintained. Since by Theorem 3.2 at most $\mathcal{O}(\log n)$ messages are sent and received in total by nodes of the same column in each round, w.h.p., and every member of v simulates at most one column of B_v, the algorithm can be executed using the global network. The runtime follows from the fact that the dimension of B_v is $\mathcal{O}(2^i)$. □

Due to Lemma 3.1, and the fact that each node simulates at most $\mathcal{O}(\log n)$ butterfly nodes, v can compute $|\Gamma(v)|$ by counting the number of remaining representatives with the Aggregate-and-Broadcast Algorithm. If $|\Gamma(v)| \geq 2^{\lceil \sqrt{2^i} \rceil}$, v becomes *successful*, in which case it does not participate in the Expansion Stage anymore. For the following, assume that v was *unsuccessful*. As we only wish to introduce *unsuccessful*

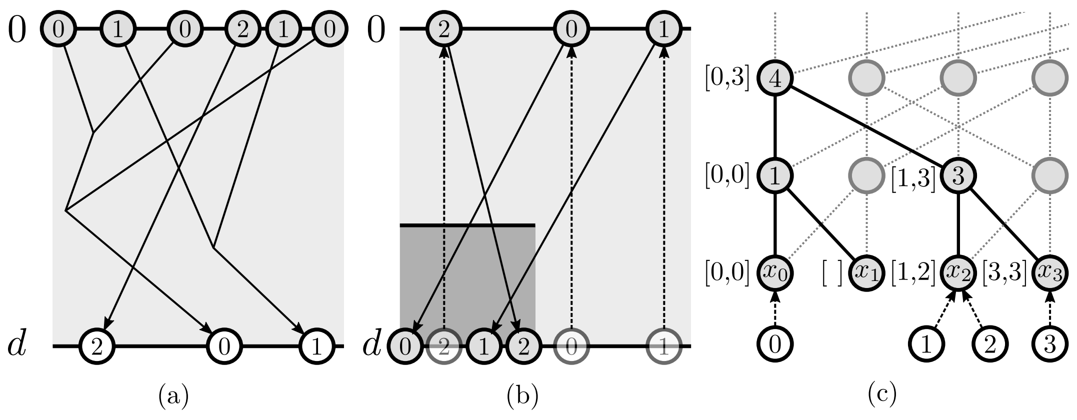


Figure 3.2.: (a) An abstract representation of B_v, where each representative stored at the top level is labeled with the identifier of its supernode. By routing all representatives of the same supernode to the same random target on the butterfly's bottom level, and combining packets whenever they reach the same node, only one representative for each supernode in $\Gamma(v)$ remains. (b) The remaining representatives are moved to random targets of b_v (dark gray). (c) Assigning unique labels from 0 to 3 to the four representatives of different supernodes (white nodes) stored in b_v. T is given by the black edges. Each node of T is labeled with an interval from which the labels of the representatives in its subtree are chosen.

supernodes to one another in each introduction step, v needs to remove all stored representatives of successful supernodes before proceeding. To do so, every member of v that stores a representative y of supernode u asks y whether its supernode u has been successful, which can be done in the global network due to Proposition 3.10. If u has been successful, y is removed. Let $\Gamma'(v) := \{u \in \Gamma(v) \mid u \text{ is unsuccessful}\}$. All remaining representatives, which are members of supernodes in $\Gamma'(v)$, are now moved into b_v by using the Route-and-Combine Algorithm as follows. Every member x of v that stores a representative y contributes $\mathrm{id}(y)$ as an input value. $\mathrm{id}(y)$ is the only value contained in routing group $R_{\mathrm{id}(y)}$, whose target is chosen uniformly and independently at random from the first $2^{2\lceil\sqrt{2^i}\rceil}$ nodes of the bottom level of B_v, which implies that the target is contained in b_v. This step is illustrated in Figure 3.2b.

Lemma 3.12 (Moving Representatives)**.** *Moving the representatives into b_v takes time $\mathcal{O}(2^i)$ and maintains Proposition 3.10, w.h.p.*

Proof. By Lemma 3.11, every node contributes at most $\mathcal{O}(\log n)$ input values, w.h.p. Furthermore, since the target of each packet is chosen uniformly and independently at random, and the total number of packets is bounded by $|\Gamma'(v)| < 2^{\lceil\sqrt{2^i}\rceil}$, using a Chernoff bound and the union bound it can easily be shown that each butterfly node is target of at most $\mathcal{O}(\log n)$ packets, w.h.p. By the same argument as in the proof of Lemma 3.11, we can perform the Route-and-Combine Algorithm in time $\mathcal{O}(2^i)$, w.h.p., using the global network. □

As the last preliminary step, every representative y of some supernode u that is stored at a member x of v needs to be replaced by a randomly chosen node that simulates a column of b_u. To do so, x sends a request to y, which is answered by a randomly chosen node of b_u with the help of the Route-and-Combine Algorithm.

More precisely, we proceed as follows. Let x be a node of v that received some requests. If x does not simulate a column of B_v, it first sends its requests to its helper node. Then, each request becomes an input value with its own routing group, whose target is a node of the bottom level of b_v chosen uniformly and independently at random. After completion of the Route-and-Combine Algorithm, every node z that simulates a column of b_v answers all requests received at its column's bottom node with a reference to itself.

Lemma 3.13 (Replacing Representatives)**.** *Replacing the representatives takes time $\mathcal{O}(2^i)$ and maintains Proposition 3.10, w.h.p.*

Proof. Clearly, each representative y of some supernode u is replaced by a randomly chosen node of b_u. It remains to show that the preconditions of Theorem 3.2 are satisfied and that Proposition 3.10 is maintained. Since Proposition 3.10 holds at the beginning of this step by Lemma 3.11, every node of B_v contributes at most $\mathcal{O}(\log n)$ input values. Furthermore, the total number of input values is bounded by $|\Gamma'(v)| < 2^{\lceil \sqrt{2^i} \rceil}$, since any supernode that stores a representative of v must be contained in $\Gamma'(v)$. Therefore, a simple application of the Chernoff bound together with the union bound implies that each node of b_v is target of at most $\mathcal{O}(\log n)$ routing groups, w.h.p. This also proves that Proposition 3.10 is maintained. □

Introduction Steps Finally, v is ready to perform the first introduction step. It will continually perform introduction steps until it either declares itself successful at the end of an introduction step, or after having performed $\lceil \sqrt{2^i} \rceil$ steps. As each introduction step takes time $\mathcal{O}(\sqrt{2^i})$, the total introduction will take time $\mathcal{O}(2^i)$.

Throughout each induction step, the sets $\Gamma(v)$ and $\Gamma'(v)$ are modified. We ensure that all representatives that are contained in $\Gamma'(v)$ at the beginning of each introduction step, and all representatives learned in this step, are stored within b_v.

Furthermore, we keep track of the distance from v to each node $u \in \Gamma(v)$ in H by storing a value $d_v(u)$. This value will help us in selecting merge edges that lead to large components of supernodes. We remark that $d_v(u)$ may not actually equal the distance $d_H(v, u)$ between u and v in H. However, it will always give the length of *some* path between u and v in H. At the beginning of the first introduction step, $d_v(u) = 1$ for all $u \in \Gamma(v)$ (and therefore also for all $u \in \Gamma'(v)$).

Proposition 3.14 (Storing Representatives)**.** *At the beginning of each introduction step, every unsuccessful supernode v stores exactly one representative y of each supernode $u \in \Gamma'(v)$. More precisely, y is a node that simulates a column of b_u, and it is stored by a node that simulates a column of b_v. Furthermore, y is annotated with $d_v(u)$.*

Clearly, the proposition holds at the beginning of the first introduction step.

To introduce all supernodes in $\Gamma'(v)$ to one another, of which there can be up to $2^{\lceil \sqrt{2^i} \rceil} - 1$ many, v's goal is to first obtain $2^{\lceil \sqrt{2^i} \rceil}$ many representatives of each supernode u. Afterwards, v will store at most $2^{2\lceil\sqrt{2^i}\rceil}$ representatives. We then compute a relation over the set of stored representatives that describes exactly which representatives are sent to one another. More precisely, we enumerate the supernodes stored in $\Gamma'(v)$ from 0 to $|\Gamma'(v)| - 1$, and the representatives of the same supernode stored by v from 0 to $2^{\lceil\sqrt{2^i}\rceil} - 1$. Let p_q be the q'th representative of the p'th supernode stored in $\Gamma'(v)$. For all p and q for which both p_q and q_p exist, v introduces q_p and p_q to one another. It is easy to see that thereby each supernode in $\Gamma'(v)$ will receive exactly one representative of every other supernode in $\Gamma'(v)$, and each representative will only be sent to at most one supernode.

To enumerate supernodes and representatives, v assigns each supernode u in $\Gamma'(v)$ a unique label $r(u) \in [|\Gamma'(v)|]$ as follows. Consider the subtree T of b_v that results from taking all paths connecting the topmost node of column 0 with all nodes on the bottom level of b_v. By performing one aggregation in T, every inner node of T learns the number of representatives stored at its leaves, and can inform its parent about this value. This allows the root of T to assign intervals of labels to its children in T, which further divide the interval according to the values received from their children, until every leaf of T that stores a representative receives unique labels for all representatives stored at it. A small example of such an assignment can be found in Figure 3.2c.

We now show how v obtains $2^{\lceil\sqrt{2^i}\rceil}$ representatives of each supernode $u \in \Gamma'(v)$. Alongside, the process will also assign each representative a of u obtained in that way a unique label $r(a) \in [2^{\lceil\sqrt{2^i}\rceil}]$. Furthermore, a will be annotated with $d_v(u)$.

In preparation for this step, each node x that simulates a column of b_v first samples $\mathcal{O}(\log n)$ many randomly chosen nodes from b_v using the Route-and-Combine Algorithm similar to Lemma 3.13. More precisely, x creates $\mathcal{O}(\log n)$ requests with random targets among the bottom nodes of b_v, and then answers every request that reach the bottom node of its column with a reference to itself.

Initially, v only stores a single representative y of each supernode $u \in \Gamma'(v)$. Let x be the node of b_v that stores the representative y of u. First, x sends a request to y that contains two of its sampled random nodes of b_v. To be able to later locally associate representatives with their supernodes, the message contains the value $r(u)$ as well as $d_v(u)$ (which x knows by Proposition 3.14). Furthermore, it contains an empty bit string l from which the representative's labels will be constructed. When y receives the request, it sends a response to both of the contained nodes, each containing one node of its set of sampled nodes of b_u. One of the two responses gets associated with label $0 \circ l$ (where $\circ$ denotes the concatenation of two binary strings), the other with $1 \circ l$, and both contain $r(u)$ and $d_v(u)$. Whenever in a subsequent round some node of b_v receives a response from u associated with label l', it sends a new request containing $r(u)$, $d_v(u)$, l', and two new random nodes of b_v, to the node of b_u that was contained in the response. In turn, every response will be answered by u with two new nodes of b_u. In the $\lceil\sqrt{2^i}\rceil$-th iteration, v receives $2^{\lceil\sqrt{2^i}\rceil}$ many representatives of u. Each representative a gets stored together with its associated label $r(a)$ and u's label $r(u)$ by its recipient in b_v.

Lemma 3.15 (Multiplying Representatives). *Every unsuccessful supernode v obtains $2^{\lceil\sqrt{2^i}\rceil}$ representatives of each supernode $u \in \Gamma'(v)$, enumerated from 0 to*

$2^{\lceil\sqrt{2^i}\rceil} - 1$, *in time* $\mathcal{O}(\sqrt{2^i})$. *Each representative of* u *is a random node of* b_u *stored at a random node of* b_v, *and is annotated by* $d_v(u)$. *Proposition 3.10 is maintained, w.h.p.*

Proof. By previous arguments and due to Theorem 3.2, sampling $\mathcal{O}(\log n)$ many random nodes at each node in b_v takes time $\mathcal{O}(\sqrt{2^i})$, w.h.p. If each node is always able answer all requests and responses sent to it, the lemma directly follows.

Therefore, it remains to show that it suffices for each node x that simulates a column of b_v to sample $\mathcal{O}(\log n)$ many random nodes of b_v. In the first request-response iteration, every representative x of b_v that stores representative y of $u \in \Gamma'(v)$ sends a request to y. By Proposition 3.10, each node has to send and receive at most $\mathcal{O}(\log n)$ requests, and therefore sends out at most $\mathcal{O}(\log n)$ responses. For these messages, a set of $\mathcal{O}(\log n)$ random nodes at each node suffices.

At the beginning of every subsequent request-response iteration, v receives at most $2^{\lceil\sqrt{2^i}\rceil}$ responses from each supernode $u \in \Gamma'(v)$, and, since $|\Gamma'(v)| < 2^{\lceil\sqrt{2^i}\rceil}$, the total number of responses received by v is $2^{2\lceil\sqrt{2^i}\rceil}$. Every response is received by a node that is chosen uniformly at random among the nodes of b_v. We enumerate *all* responses received by v throughout all $\lceil\sqrt{2^i}\rceil$ iterations, fix some node x of b_v, and define X_i as the binary random variable that is 1 if and only if the i-th response is received by x. Let X be the sum of all X_i. Since b_v is simulated by $2^{2\lceil\sqrt{2^i}\rceil}$ nodes, $\mathbb{E}[X] \leq \lceil\sqrt{2^i}\rceil = \mathcal{O}(\log n)$, and we can apply a Chernoff bound and the union bound to show that x will receive at most $\mathcal{O}(\log n)$ responses over the course of all iterations, w.h.p.

Since $|\Gamma(v)| < 2^{\lceil\sqrt{2^i}\rceil}$, by Proposition 3.9 at most $2^{\lceil\sqrt{2^i}\rceil}$ supernodes may store a representative of v. Therefore, we can symmetrically argue that the total number of requests received by v in total is $\lceil\sqrt{2^i}\rceil \cdot 2^{2\lceil\sqrt{2^i}\rceil}$. Since each request is sent to a node chosen uniformly at random from b_v, it analogously follows that every node x of b_v will receive at most $\mathcal{O}(\log n)$ requests over the course of all iterations, w.h.p. Therefore, it suffices for each node to sample $c \log n$ many nodes, where c is a constant prescribed from the desired success probability. ☐

We are now ready to perform the actual introduction. Following our notation from earlier, we denote the representative y of supernode $u \in \Gamma'(v)$ such that $r(u) = p$ and $r(y) = q$ as p_q. The goal of v is to send p_q as well as $d_v(p) + d_v(q)$ to q_p for all q, p for which both p_q and q_p exist. To do so, one node of b_v needs to learn all these values and take care of the introduction. More specifically, if $p \leq q$, then p_q is moved from the top node of x's column to the bottom node of column $h(p, q)$, where $h : [2^{\lceil\sqrt{2^i}\rceil}]^2 \to [2^{2\lceil\sqrt{2^i}\rceil}]$ is a (pseudo-)random hash function known by all nodes; if $p > q$, then p_q is moved to column $h(q, p)$ instead. This is done with an application of the Route-and-Combine Algorithm similar to Lemma 3.12. If q_p exists, then either the bottom node of $h(p, q)$ (if $p \leq q$), or the bottom node of $h(q, p)$ (if $p > q$) receives both p_q and q_p, which can then send p_q annotated by $d_v(p) + d_v(q)$ to q_p.

The following lemma summarizes the outcome of each introduction step. It follows from the fact that we introduce random representatives of unsuccessful supernodes such that each representative stored at v is sent at most once, and each supernode in $\Gamma'(v)$ receives one random representative of any other supernode in $\Gamma'(v)$.

Lemma 3.16 (Introducing Supernodes)**.** *Let $u \in \Gamma'(v)$ and $w \in \Gamma'(u)$ be unsuccessful supernodes. v receives a representative of w from u, which is annotated with the value $d_u(v) + d_u(w)$. More precisely, the representative is a node of b_w that gets stored at a node of b_v. The introduction takes time $\mathcal{O}(\sqrt{2^i})$, w.h.p. Proposition 3.10 is maintained, w.h.p.*

Proof. Clearly, v receives a representative of w from u as claimed in the lemma. Since Proposition 3.10 holds before the introduction by Lemma 3.15, every node contributes at most $\mathcal{O}(\log n)$ values to the Route-and-Combine Problem, w.h.p, each having its own routing group. Furthermore, by using the Chernoff and union bound, it can easily be shown that each node is target of at most $\mathcal{O}(\log n)$ routing groups, w.h.p. Since b_v has dimension $\mathcal{O}(\sqrt{2^i})$ and by Theorem 3.2, applying the Route-and-Combine Algorithm takes time $\mathcal{O}(\sqrt{2^i})$, w.h.p. This also implies that Proposition 3.10 is maintained, w.h.p. □

At this point, the set of supernodes discovered by v is extended to

$$\Gamma(v) \leftarrow \Gamma(v) \cup \bigcup_{u \in \Gamma'(v)} \Gamma'(u).$$

However, the nodes of b_v may still store representatives of the *same* supernode u, which happens if at least two nodes in $\Gamma'(v)$ introduce u to v. We now filter these representatives such that exactly one representative is stored for each supernode. Specifically, we want to store the representative with smallest annotated distance value. This can be done by performing the Route-and-Combine Algorithm as in Lemma 3.11, but in b_v only. By choosing the aggregate function f as the minimum function MIN, only the representative with smallest distance value of each supernode in $\Gamma(v)$ remains. Since Proposition 3.10 holds by the previous lemma, this can be done in time $\mathcal{O}(\sqrt{2^i})$, and maintains the proposition, w.h.p. Finally, v counts the number of representatives now stored in b_v using the Aggregate-and-Broadcast Algorithm to determine $|\Gamma(v)|$. If $|\Gamma(v)| \geq 2^{\lceil \sqrt{2^i} \rceil}$, becomes *successful*. Irrespectively, v declares itself successful if one of its neighbors has become successful in the previous introduction step, or was successful already before the first introduction step. Finally, every member of v that stores a representative x asks x whether its supernode u is now successful, which can be done in the global network due to Proposition 3.10. If u is successful, x is removed from b_v, and $\Gamma'(v)$ becomes the set of remaining unsuccessful supernodes in $\Gamma(v)$.

To complete our analysis of the introduction step, we conclude the following.

Lemma 3.17 (Conclusion of the Introduction Step)**.** *The introduction step takes time $\mathcal{O}(\sqrt{2^i})$. Propositions 3.9, 3.10, and 3.14 are maintained for the next introduction step, w.h.p.*

End of the Expansion Stage All supernodes stop after having performed at most $\lceil \sqrt{2^i} \rceil$ introduction steps. If a supernode is unsuccessful until the end of the last introduction step, but determines that one of its neighbors has become successful in this step, it also declares itself successful. Note that at the end of the Expansion Stage, a supernode may still be unsuccessful, and even if is has been successful, it might have learned fewer than $2^{\lceil\sqrt{2^i}\rceil}$ many supernodes (i.e., when it became successful because of one of its neighbors).

Recall that H is the graph of adjacent active supernodes as defined at the beginning of Section 3.4.2. We make the following observation for the outcome of the expansion stage.

Lemma 3.18 (Outcome of Expansion Stage)**.** *Let C be a connected component of H. If $|C| \geq 2^{\lceil\sqrt{2^i}\rceil}$, then every supernode in C is successful at the end of the Expansion Stage. Otherwise, no supernode in C is successful.*

Proof. First, if $|C| < 2^{\lceil\sqrt{2^i}\rceil}$, then for every $v \in C$ we have $|\Gamma(v)| < 2^{\lceil\sqrt{2^i}\rceil}$ after each introduction step, since any supernode can only discover supernodes of C. Therefore, no supernode in C will ever declare itself successful.

Now let $|C| \geq 2^{\lceil\sqrt{2^i}\rceil}$ and consider the hypothetical execution in which the supernodes can perform unlimited communication and unceasingly continue to introduce discovered nodes without declaring themselves successful. We show by induction on the number of introduction steps that in such an execution $\Gamma(v)$ of every supernode v of C contains all supernodes within 2^t hops in H after introduction step t.

Before step 1, $\Gamma(v)$ contains all neighbors of v in H. Since none of these supernodes will initially be successful by assumption, v discovers all neighbors of its neighbors in H in the first introduction step . Assume that $\Gamma(v)$ contains every supernode within 2^t hops in H before introduction step $t+1$ for all v in C. Fix some v, and let $w \notin \Gamma(v)$ be a supernode within 2^{t+1} hops from v in H. Then there must be a supernode $u \in \Gamma(v)$ such that w lies within 2^t hops from u, and, consequently, $w \in \Gamma(u)$. Since $v \in \Gamma(u)$ by Proposition 3.9, and v, u and w are unsuccessful by our assumption, u introduces w to v in introduction step $t+1$. We conclude that after $\lceil\sqrt{2^i}\rceil$ introduction steps, $\Gamma(v)$ contains all supernodes within distance $2^{\lceil\sqrt{2^i}\rceil}$, which must be at least $2^{\lceil\sqrt{2^i}\rceil}$ many supernodes.

Therefore, the only way for a supernode v in C to *not* be successful after the last introduction step is if some supernode in C becomes successful and is subsequently not introduced any further, leading to v not discovering sufficiently many supernodes. Assume that this is the case, and compare the actual execution in which v does not discover $2^{\lceil\sqrt{2^i}\rceil}$ many supernodes due to some supernode becoming successful with the hypothetical execution described in the previous paragraph. Then, there must be a first introduction step t in which v does *not* learn some supernode u in the actual execution that it *would* learn if no supernode ever became successful. We show by induction on t that v will declare itself successful at the end of introduction step $\min\{t+1, \lceil\sqrt{2^i}\rceil\}$.

First, let $t = 1$. Since v does not learn supernode u, some node $w \in \Gamma(v)$ must have removed its representative of u because u was already successful before the first introduction step. Thus, w declares itself successful after the first step, leading to u becoming successful at latest after the second introduction step.

Now let $t > 1$. By the same reasoning, there must be a node $w \in \Gamma(v)$ at the beginning of introduction step t that does not introduce some supernode u to v in introduction step t. This is because either u was successful before introduction step t, or because u was not introduced to w in some step $t' < t$. In the first case, w

becomes successful after step t, since $u \in \Gamma(w)$ and u became successful before step t. In the second case, the inductive hypothesis implies that w becomes successful after step $t' + 1 \leq t$ as well. If $t < \lceil\sqrt{2^i}\rceil$, v becomes successful after step $t + 1$. If otherwise t was the last introduction step, v will declare itself successful after w does so. $\square$

The final lemma of this section follows from Lemma 3.17 and the fact that we perform $\lceil\sqrt{2^i}\rceil$ introduction steps.

Lemma 3.19 (Expansion Stage Runtime)**.** *The Expansion Stage takes time $\mathcal{O}(2^i)$ and can be performed using global communication, w.h.p.*

3.4.3. Merging Stage

After having collected potentially large sets of neighbors in the Expansion Stage, the goal of each active supernode in the Merging Stage is to select merge edges from $G_2 = (V, E_2)$ that connect members of supernodes. We will ensure that the total number of members contained in each component of supernodes connected by merge edges (which we call *merge component*) amounts to at least $2^{2^i+(j+1)\lceil\sqrt{2^i}\rceil}$. Therefore, each resulting supernode u will be sufficiently large to ensure Proposition 3.6 for the beginning of the next subphase. Additionally, the selected merge edges connect the internal spanning trees of all supernodes within a merge component to a tree that contains all nodes of the resulting supernode. This will maintain Proposition 3.7. Finally, towards Proposition 3.8, we need to show how to establish the internal butterfly of each supernode that is active in the next subphase.

Let v be a supernode and C_v be the component of H that contains v. By Lemma 3.18, we know that the supernodes within C_v are either all successful or all unsuccessful. For the selection of merge edges, we consider these cases separately.

Successful Supernodes: Grouping Step 1 First, let v be a successful supernode. We define $t(v) = i$ if v became successful at the end of introduction step $i \in \{1, \ldots, \lceil\sqrt{2^i}\rceil\}$. Further, we define $t(v) = 0$, if v was successful at the beginning already, and $t(v) = \lceil\sqrt{2^i}\rceil + 1$ if v became successful after the last introduction step due to a neighbor u that became successful (for which $t(u) = \lceil\sqrt{2^i}\rceil$). Furthermore, let $\phi(v) := t(v) \circ \mathrm{id}(v)$. Note that $\phi(u) < \phi(v)$ if and only if $t(u) < t(v)$, or $t(u) = t(v)$ and $\mathrm{id}(u) < \mathrm{id}(v)$. Let $m(v) = \mathrm{argmin}_{u \in \Gamma(v)} \phi(u)$.

Successful supernodes select merge edges in two *grouping steps*, which we refer to as Grouping Step 1 and Grouping Step 2. For Grouping Step 1, we need the following observation.

Lemma 3.20. *If $\phi(m(v)) < \phi(v)$, then v is adjacent to a supernode u in H such that $m(v) \in \Gamma(u)$ and $d_u(m(v)) < d_v(m(v))$.*

Proof. We define the *introduction path* $I = (v = v_0, v_1, \ldots, v_k = m(v))$ from v to $m(v)$, where v_1 is adjacent to v in H, and v_{t+1} has been introduced to v by v_t for all $1 \leq t \leq k - 1$. Note that multiple nodes may introduce v_{t+1} to v. For any t, we always choose v_t as the *first* node that introduces v_{t+1} to v. If there are multiple such nodes, then we let v_t be the node such that $d_v(v_t) + d_{v_t}(v_{t+1})$ is minimized, breaking ties arbitrarily. For any $1 \leq t \leq k-1$, let $r_t \geq t$ be the introduction step in which v learned v_{t+1} from v_t. Clearly, $r_t < r_{t+1}$ for all $1 \leq t \leq k-2$. Since the distance values $d_v(v_{t+1})$ always correspond to lengths of actual paths in H, it is impossible for v to learn v_{t+1} in introduction step r_t, and obtain a smaller distance value in a subsequent introduction step. Therefore, for our choice of the nodes of I we have that $d_v(v_{t+1}) = d_v(v_t) + d_{v_t}(v_{t+1})$ for all $1 \leq t \leq k-1$.

We show by induction on t that (1) at the beginning of introduction step r_t, $v_1 \in \Gamma'(v_t)$, for all $1 \leq t \leq k-1$, and (2) $d_{v_1}(v_t) + 1 \leq d_v(v_t)$ for all $1 \leq t \leq k$. For $t = 1$, both statements hold since v_2 is adjacent to v_1, and $d_{v_1}(v_1) + 1 = d_v(v_1) = 1$ at the beginning of introduction step r_1. Now assume that the induction hypothesis holds for some $t \leq k-1$. Therefore, $v_1 \in \Gamma'(v_t)$, and $d_{v_1}(v_t) + 1 \leq d_v(v_t)$. Since $m(v)$ eventually has the minimum ϕ-value among all supernodes of $\Gamma(v)$ and becomes successful no earlier than at the end of introduction step r_{k-1}, no node $v_{t'}$ for $t' \leq t+1$ can be successful before introduction step r_t, as otherwise v would know a node with smaller ϕ-value.

Therefore, v_t will introduce v_1 to v_{t+1} in introduction step r_t. If $t+1 \leq k-1$, then v_1 will also not be finished at the end of introduction step r_t, so v_1 will be included in $\Gamma'(v_{t+1})$ at the beginning of the introduction step r_{t+1}, which establishes Induction Hypothesis (1). Additionally, even if $t+1 = k$, we have that

$$\begin{aligned} d_v(v_{t+1}) &= d_v(v_t) + d_{v_t}(v_{t+1}) \\ &\geq d_{v_1}(v_t) + 1 + d_{v_t}(v_{t+1}) \\ &\geq d_{v_1}(v_{t+1}) + 1, \end{aligned}$$

where the first equality holds by our choice of v_t, the second inequality holds due to Induction Hypothesis (2), and the third inequality holds since v_1 will receive $d_{v_t}(v_{t+1})$ from v_t. □

By the previous lemma, v can select a merge edge according to the following rule.

Grouping Step 1: If $\phi(m(v)) < \phi(v)$, then v selects a merge edge to a supernode $u \in \Gamma(v)$ adjacent to v in H such that $m(v) \in \Gamma(u)$ and $d_u(m(v))$ is minimized. More precisely, the merge edge that v adds to $\mathcal{E}$ is an edge $\{x, y\} \in E_2$, where x is a member of v adjacent to a member y of u.

An example of the selection of merge edges in Grouping Step 1 can be found in Figure 3.3a. Clearly, v knows id(u) for all $u \in \Gamma(v)$, and, since each u informed v if it became successful before or at the end of the last introduction step, can infer $t(u)$. Therefore, each member of v that stores a representative of u knows $\phi(u)$. By using the Aggregate-and-Broadcast Algorithm, the members of v can easily compute $m(v)$ and $\phi(m(v))$ in time $\mathcal{O}(2^i)$, and determine whether $\phi(m(v)) < \phi(v)$. In this case, v needs to communicate with its adjacent supernodes in H to determine which of those supernodes minimizes $d_u(m(v))$.

Let x be a member of v that stores a representative y of an adjacent supernode u after filtering the representatives before the first introduction step (see Lemma 3.11); those are the supernodes adjacent to v in H. x sends a request to y and asks for $d_u(m(v))$. The requests are processed with an application of the Route-and-Combine

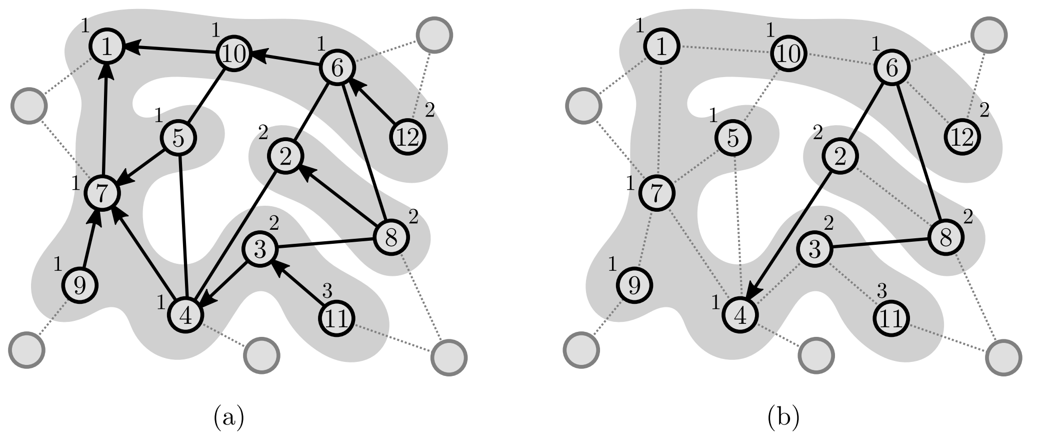


Figure 3.3.: (a) Active successful supernodes select merge edges in Grouping Step 1. Each active supernode is labeled with its ϕ-value (inactive supernodes are gray). The small number at the side of each supernode v indicates $\phi(m(v))$; in this example, $\Gamma(v)$ contains all active supernodes within hop-distance 2 in H. The selection of merge edges results in two components of supernodes (gray shade). (b) The smaller of the two components did not grow large enough, and selects a merge edge in Grouping Step 2. As a result, both components are merged into one large supernode.

Algorithm. More precisely, all requests at u that ask for $d_u(w)$ for some w are in routing group $R_{\mathrm{id}(w)}$ and are routed to target $t_{\mathrm{id}(w)}$ on the bottom level of B_u chosen uniformly at random using a (pseudo-)random hash function $h : \{0,1\}^* \to [2^{\lfloor \log |u| \rfloor}]$. Additionally, we let $t_{\mathrm{id}(w)}$ learn $d_u(w)$ by letting the member of u that stores $d_u(w)$ directly send a packet to $t_{\mathrm{id}(w)}$ (clearly, if no such member exists, $t_{\mathrm{id}(w)}$ does not receive any distance value). To answer the requests, the path system formed by routing all packets to $t_{\mathrm{id}(w)}$ (which basically forms a binary tree in B_u) is used in reverse direction to inform all members of u that initially received a request for $d_u(w)$. To be able do so, each butterfly node needs to record all packets that are routed through it during the execution of the Route-and-Combine Algorithm. It immediately follows from the proof of Theorem 3.2 that the recipients of requests for $d_u(w)$ learn this value in time $\mathcal{O}(2^i)$, w.h.p., and can then answer all requests directly.

After having received all responses, v determines the supernode u adjacent to it that minimizes $d_u(m(v))$ with the Aggregate-and-Broadcast Algorithm. Similarly, the members of v can learn the edge $\{x, y\} \in E_2$ such that x is a member of v and y is the member of u and $(\mathrm{id}(x), \mathrm{id}(y))$ is minimized, which is then selected as the merge edge. This implies the following lemma.

Lemma 3.21 (Runtime of Grouping Step 1)**.** *Selecting merge edges in Grouping Step 1 takes time $\mathcal{O}(2^i)$, w.h.p.*

Let H_{merge} be the graph that contains all successful supernodes and a directed edge (u, w) if u selected a merge edge to w, and let $C_{v'}$ be the component of H_{merge} that contains v; the supernodes of $C_{v'}$ will merge to form a new supernode v'. Furthermore, let $E_{v'} \subseteq E_2$ be the set of merge edges selected by the supernodes of $C_{v'}$. For the outcome of Grouping Step 1, have the following lemma (recall that $\mathcal{E}_u$ is the internal spanning tree of each supernode u).

Lemma 3.22 (Outcome of Grouping Step 1)**.** *$C_{v'}$ is a rooted tree whose root has the smallest ϕ-value among all supernodes of $C_{v'}$. Furthermore, $\mathcal{E}_{v'} := E_{v'} \cup \bigcup_{u \in C_{v'}} \mathcal{E}_u$ is a tree that contains all members of v'.*

Proof. To show that $C_{v'}$ forms a rooted tree, first note that every supernode selects at most one merge edge, therefore every supernode of $C_{v'}$ has outdegree at most 1. It remains to show that $C_{v'}$ does not contain a directed cycle. Assume that supernode u of $C_{v'}$ selected a merge edge to $w \in \Gamma(u)$. Therefore, $m(u) \in \Gamma(w)$ and thus $\phi(m(w)) \leq \phi(m(u))$. We have that either $\phi(m(w)) < \phi(m(u))$, or $\phi(m(w)) = \phi(m(u))$, which implies that $m(w) = m(u)$, but $d_w(m(w)) < d_u(m(u))$ by Lemma 3.20. Therefore, if we label each supernode u with $l(u) := (\phi(m(u)), d_u(m(u)))$ and consider the lexicographic order, then $l(u) > l(w)$ for each edge (u, w) in $C_{v'}$. Furthermore, since the $\phi(m(u))$-values monotonously decrease along a directed path, and $m(r) = r$ for the root r of $C_{v'}$, r has the smallest ϕ-value among all supernodes of $C_{v'}$.

Since by Proposition 3.7 the internal spanning tree $\mathcal{E}_u$ of each supernode u is a tree, the union of all internal trees of $C_{v'}$ together with the merge edges $E_{v'}$ form a tree that contains all members of v'. □

At the end of Grouping Step 1, our goal is to construct the internal butterfly of each resulting supernode. To do so, we present a variation of the merging step of the *Overlay Construction Algorithm* of Gmyr et al. [Gmy+17a]. The algorithm is based on the well-known *Euler tour* technique (e.g., [TV85; AV84]), which basically applies *pointer jumping* [JáJ92] to trees. Since we do not know exactly how large the resulting supernodes will be, we control the runtime of our algorithm with a parameter k. The algorithm may not finish in time within a supernode that is too large; in this case, the supernode will simply become inactive. Formally, we prove the following lemma.

Lemma 3.23 (Merging)**.** *Let T be an (undirected) tree with degree $\mathcal{O}(\log n)$ that contains N nodes. There is an algorithm that, if $k \geq \log(N-1) + 1$ for some parameter $k = \mathcal{O}(\log n)$, takes $\mathcal{O}(k)$ rounds to let every node of T learn N and the highest identifier of all nodes in T. Furthermore, the algorithm arranges the nodes as a path graph $L = (x_0, \dots, x_{N-1})$ that contains the nodes in the order in which they are visited in some depth-first traversal of T starting at the node with highest identifier. If $k < \log N + 1$, the algorithm fails. After termination, all nodes know whether the algorithm succeeded or not.*

Proof. Our algorithm is divided into the following three steps.

- **Step 1:** T is transformed into a directed cycle graph C of *virtual nodes*, in which each node x of T simulates $\deg_T(x) = \mathcal{O}(\log n)$ many virtual nodes.
- **Step 2:** C is transformed into a path graph $L = (x_0, \dots, x_{N-1})$ that contains the nodes of T in the order in which they are visited in some depth-first traversal of T starting at the node with highest identifier.

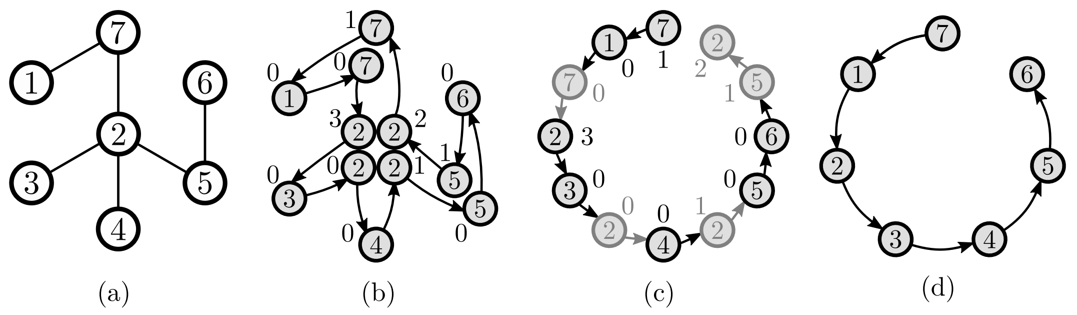


Figure 3.4.: (a) A tree T with node identifiers. (b) Each node x simulates $\deg_T(x)$ many virtual nodes that are connected according to a depth-first traversal of T. In addition to the identifier of its original node x, each virtual node x^l is annotated with its index l (small numbers at the side). (c) The path graph L' that begins at the node with highest identifier, which in this case is 7.1. All virtual nodes that are not marked (light gray) will be removed from the path graph. (d) The path graph L that results from compressing L' to its marked nodes and replacing them by the corresponding nodes of T.

- **Step 3:** Each node x_l in L learns $x_{l'}$ for all $l' = l \pm 2^t$, $t \in N_0$, if $x_{l'}$ exists. Furthermore, x_i learns N and the highest identifier of all nodes in T.

Clearly, a successful outcome of Step 3 implies the lemma. We will make sure that all nodes terminate in the same round after having performed $\mathcal{O}(k)$ rounds, and can then easily infer whether the algorithm was successful.

Step 1 For Step 1, we denote the neighbors of each node x in T by ascending identifier as $x(0), \dots, x(\deg(x)-1)$. Consider the (cyclic) depth-first traversal in T that first traverses an arbitrary edge of T, and, when visiting x from $x(l)$, continues at x's neighbor $x((l+1) \bmod \deg(x))$. Intuitively, C is the directed cycle graph that corresponds to this traversal. More specifically, every node x simulates the nodes $x^0, \dots, x^{\deg_T(x)-1}$, where x^l corresponds to the traversal visiting x from $x(l)$. The cycle graph C contains all virtual nodes and an edge $(x^l, y^{l'})$ in C for all $x^l, y^{l'}$ such that $y = x((l+1) \bmod \deg_T(x))$ and $x = y(l')$ (see Figure 3.4a and 3.4b for an example). To accordingly introduce each virtual node to its predecessor in C, every node x sends the *virtual identifier* $\mathrm{id}(x^l) := \mathrm{id}(x) \circ l$ to $x(l)$ for all $l \in [\deg_T(x)]$. Since T has degree $\mathcal{O}(\log n)$, each node only has to simulate $\mathcal{O}(\log n)$ virtual nodes. Furthermore, since there are two edges in C for each edge of T, $|C| = 2(N-1)$.

Step 2 To transform C into a list L that contains every node of T, we perform pointer jumping in C to establish *shortcut edges*. Every virtual node x first selects its successor in C as its *left neighbor* ℓ_1, and its predecessor as its *right neighbor* r_1. In the first round, every virtual node x establishes $\{\ell_1, r_1\}$ as a new shortcut edge by sending the edge to both ℓ_1 and r_1. Whenever at the beginning of some round $t > 1$ x receives shortcut edges $\{y, x\}$ and $\{x, z\}$ from ℓ_{t-1} and r_{t-1}, respectively, it sets $\ell_t := y$, $r_t := z$ and establishes $\{\ell_t, r_t\}$ by informing ℓ_t and r_t. Note that a shortcut edge established in round t bridges 2^t many virtual nodes in C (i.e., there is a path with 2^t hops from x to ℓ_{t+1} and r_{t+1}). After having performed k rounds, where k is the prescribed runtime parameter, the nodes stop.

Now, the nodes try to determine the virtual node x^* that has the highest virtual identifier, as well as the length $d^*(x)$ of the directed path from x^* to each virtual node x in C. If k has not been chosen sufficiently large, this step will fail, which will be noticed by all nodes. As a first step, every virtual node x sends a message $(\mathrm{id}(x), 0)$ to itself. Then, in round $t = 1, \dots, k$, each virtual node selects the message $(\mathrm{id}(y), d)$ with largest identifier received so far, and sends $(\mathrm{id}(y), d + 2^{t-1})$ to ℓ_t. It is easy to see that after round t, the identifier of x^* has reached all nodes within $2^t - 1$ many hops from x^* in the directed cycle. Therefore, if $k \geq \log(N-1) + 1 = \log(2(N-1)) = \log|C|$, after k rounds all virtual nodes store x^*. In this case, x^* determines that it is the virtual node with highest identifier by looking at its predecessor in C, and informs all virtual nodes using the shortcut edges as before within an additional k rounds. Conversely, if the predecessor of C did not receive x^*, we must have that $k < \log|C|$. In this case, x^* will not inform all nodes, and after having received no notification within k rounds, the nodes declare the execution as unsuccessful.

Assume that the nodes were successful, i.e., $k \geq \log(N-1) + 1$. By letting x^* remove the edge to its predecessor in C, we obtain a path graph L' that begins at x^* and contains all virtual nodes (see Figure 3.4c). Each node x of T now *marks* its virtual node y for which $d^*(y)$ is minimal. Let U be a maximal subgraph of L' that consists of unmarked nodes only. U is adjacent to at most 2 marked nodes, and we can easily introduce these nodes by performing pointer jumping on the unmarked nodes for k rounds. After removing all unmarked nodes and replacing each marked node with its corresponding original node, we obtain a path graph $L = (x_0, \dots, x_{N-1})$ that contains all nodes of T (see Figure 3.4d). Furthermore, since we mark only the nodes that appear *first* when traversing C from starting at x^*, and C corresponds to a depth-first traversal of T, L contains the nodes in the order in which they are visited in a depth-first traversal starting at the node with highest identifier.

Step 3 By performing pointer jumping on L for k rounds, every node x_l learns $x_{l'}$ for $l' = l \pm 2^t$, $t \in [k+1]$, if that node exists. By performing a broadcast from x_0, which is the node with highest identifier among all nodes of T, the nodes learn $\mathrm{id}(x_0)$ and N. More precisely, x_0 first sends a message $(\mathrm{id}(x_0), 1 + 2^l)$ to x_{2^l} for all $l \in [k+1]$, if that node exists. Whenever node x_t receives $(\mathrm{id}(x_0), d)$, it sends $(\mathrm{id}(x_0), d + 2^t)$ to x_{t+2^l} for all $l \in [k+1]$, if it exists. After k rounds, x_{N-1} knows both $\mathrm{id}(x_0)$ and N, which is then sent to all nodes with a final broadcast. $\square$

To construct the internal butterfly of supernode v' from Lemma 3.22, we use the algorithm of Lemma 3.23 on $\mathcal{E}_{v'}$. By Lemma 3.22, $\mathcal{E}_{v'}$ is a tree that contains all members of v', and since G_2 has degree $\mathcal{O}(\log n)$, $\mathcal{E}_{v'}$ has degree $\mathcal{O}(\log n)$ as well. By choosing $k = 2^i + (j+2)\lceil\sqrt{2^i}\rceil + 1 = \mathcal{O}(2^i)$, the algorithm is guaranteed to succeed in v' if v' contains at most $2^{2^i + (j+2)\lceil\sqrt{2^i}\rceil}$ nodes. In that case, all members of v' learn whether $|v'| \leq 2^{2^i + (j+2)\lceil\sqrt{2^i}\rceil} - 1$, in which case v' is supposed to be active in the next subphase. If the algorithm does not succeed, v' must be too large anyway, and all nodes can infer that v' will be inactive.

Successful Supernodes: Grouping Step 2 If $|v'| \leq 2^{2^i+(j+2)\lceil\sqrt{2^i}\rceil} - 1$, we might have that v' is too small for the next subphase, which is the case if $|v'| < \min\{2^{2^i+(j+1)\lceil\sqrt{2^i}\rceil}, n\}$. To resolve this problem, the goal of Grouping Step 2 is to let v' pick an additional merge edge such that the resulting supernodes are of sufficient size.

To avoid confusion with the supernodes before Grouping Step 1, in the following we call each supernode that resulted from supernodes that merged in Grouping Step 1 an *intermediate supernode*. For each intermediate supernode u', we define $\phi(u')$ as the ϕ-value of the root of $C_{u'}$ (which is the rooted tree of supernodes that merge to form u', see Lemma 3.22). Before we describe how merge edges are selected in Grouping Step 2, we make the following observation.

Lemma 3.24. *Let v' be an intermediate supernode, and let*

$$|v'| < \min\{2^{2^i+(j+1)\lceil\sqrt{2^i}\rceil}, n\}.$$

v' is adjacent in G_2 to an intermediate supernode u' such that $\phi(u') < \phi(v')$.

Proof. v' is the result of merging all supernodes of the rooted tree $C_{v'}$. Let v be the root of that tree. Since v did not select any merge edge, we have that $\phi(v) = \phi(m(v))$. This implies that v did not become successful in the Expansion Stage because of any supernode $u \in \Gamma(v)$; otherwise, we would have $\phi(u) < \phi(v)$ and $\phi(m(v)) \leq \phi(u) < \phi(v)$. Since v became successful anyway, we must have that $|\Gamma(v)| \geq 2^{\lceil\sqrt{2^i}\rceil}$. By Proposition 3.6, each supernode $u \in \Gamma(v)$ is of size $|u| \geq \min\{2^{2^i+j\lceil\sqrt{2^i}\rceil}, n\}$. If all these nodes were now contained in v', then $|v'| \geq \min\{2^{2^i+(j+1)\lceil\sqrt{2^i}\rceil}, n\}$, which contradicts the assumption of the lemma. Therefore, there must be a supernode $u \in \Gamma(v)$ that is not contained in $C_{v'}$.

First, consider the case that there exists such a supernode u that was adjacent in H to some $w \in C_{v'}$ before Grouping Step 1. Since $v \in \Gamma(u)$ by Proposition 3.9, we have that $\phi(m(u)) \leq \phi(v)$. By Lemma 3.22, for the intermediate supernode u' that u merges into we have that $\phi(u') \leq \phi(m(u)) \leq \phi(v')$. More specifically, since $u' \neq v'$, which is due to the fact that u is not contained in $C_{v'}$, we have that $\phi(u') < \phi(v')$, proving the claim.

In the other case, every supernode $u \in \Gamma(v)$ that is not contained in $C_{v'}$ is also not adjacent to any supernode in $C_{v'}$. Let P be a shortest path from v to such a supernode u in H. Let w be the first node on that path from v to u that is not contained in $C_{v'}$. Since u was introduced to v, but w was *not*, w must have been successful before v. Therefore, $\phi(m(w)) \leq \phi(w) < \phi(v) = \phi(v')$. Let w' be the intermediate supernode into which w merges. From Lemma 3.22 we conclude that $\phi(w') \leq \phi(m(w)) < \phi(v')$. □

In Grouping Step 2, every intermediate node that is still too small selects an additional edge to an adjacent intermediate supernode w'. We will show that the resulting supernode will be sufficiently large. Formally, intermediate supernodes select merge edges according to the following rule.

Grouping Step 2: Let v' be an intermediate supernode for which $|v'| < \min\{2^{2^i+(j+1)\lceil\sqrt{2^i}\rceil}, n\}$. v' selects a merge edge to an intermediate supernode u' adjacent to v' in G_2 such that $\phi(u') < \phi(v')$. More precisely, the merge edge that v' adds to $\mathcal{E}$ is an edge $\{x, y\} \in E_2$, where x is a member of v' adjacent to a member y of u'.

An example of the selection of merge edges in Grouping Step 2 can be found in Figure 3.3b.

To select a viable supernode u', we remark that v' does not have to know $\phi(u')$. In fact, if the application of Lemma 3.23 was not successful, which happens if u' is already too large, then the members of u' themselves do not even learn $\phi(u')$. However, it suffices to determine a supernode u for which $\phi(m(u)) \leq \phi(v')$ that is adjacent to one of the supernodes of $C_{v'}$, but not contained in $C_{v'}$. The existence of such a supernode follows from the proof of Lemma 3.24. Therefore, we can select a merge edge by letting each member x of v' ask each of its neighbors y in G_2, which was previously contained in some supernode u, for $\phi(m(u))$, and compute the minimum using the Aggregate-and-Broadcast Algorithm in time $\mathcal{O}(2^i)$. This yields the following lemma.

Lemma 3.25 (Runtime of Grouping Step 2)**.** *Selecting merge edges in Grouping Step 2 takes time $\mathcal{O}(2^i)$, w.h.p.*

Similar to Lemma 3.22, for each resulting supernode v we define C_v as the component of v' in the directed graph that contains an edge (u', w') if u' selected a merge edge to w' in Grouping Step 2. Furthermore, let $E_v \subseteq E_2$ be the set of merge edges selected by the intermediate supernodes of C_v. The outcome of Grouping Step 2 can be summarized as follows.

Lemma 3.26 (Outcome of Grouping Step 2)**.** *Each resulting supernode v has size $|v| \geq \min\{2^{2^i+(j+1)\lceil\sqrt{2^i}\rceil}, n\}$. Furthermore, C_v is a directed tree and $\mathcal{E}_v := E_v \cup \bigcup_{u' \in C_v} \mathcal{E}_{u'}$ is a tree that contains all members of v.*

Proof. If v' chooses to merge with u', then $\phi(u') < \phi(v')$, which implies that there cannot be directed cycles in C_v; therefore, C_v is a directed tree. Since each intermediate supernode u' is internally organized as a tree $\mathcal{E}_{u'}$ by Lemma 3.22, we have that $\mathcal{E}_v$ is a tree that contains all members of v.

For the first claim of the lemma, note that the root u' of C_v did not select any merge edge. Since by Lemma 3.24 it can always select such an edge if it is not of sufficient size, the only reason for not choosing an edge must be that it is already large enough. More precisely, $|u'| \geq \min\{2^{2^i+(j+1)\lceil\sqrt{2^i}\rceil}, n\}$, which directly implies the lemma. □

To merge each supernode that results from Grouping Step 2, we can therefore again use the algorithm of Lemma 3.23 with $k = 2^i + (j+2)\lceil\sqrt{2^i}\rceil + 1$. We conclude the following lemma for all supernodes that result from merging successful supernodes in the Merging Stage.

Lemma 3.27 (Merging Successful Supernodes)**.** *Merging successful supernodes takes time $\mathcal{O}(2^i)$, w.h.p. Propositions 3.6, 3.7, and 3.8 are maintained for the next subphase.*

Unsuccessful Supernodes It remains to show how the unsuccessful supernodes attain a sufficient size. Similar to successful supernodes, every unsuccessful supernode will select at most one merge edge, which forms clusters of supernodes that we attempt to merge using the algorithm of Lemma 3.23. However, this time only a *single* grouping step will suffice for the resulting supernodes to be sufficiently large.

Let v be an unsuccessful supernode. Since v is unsuccessful, by Lemma 3.18 the component C of H that contains v must consist of fewer than $2^{\lceil\sqrt{2^i}\rceil}$ supernodes, all of which being unsuccessful. From the proof of the lemma we also infer that $\Gamma(v)$ contains all supernodes in C. Furthermore, if there still exists an inactive supernode, one of the supernodes of C must be adjacent to an inactive supernode in G_2. Conversely, if none is adjacent to an inactive supernode anymore, C must contain all remaining supernodes. In the first case, we let a supernode u^* of C merge with an inactive supernode w that u^* is adjacent to in G_2, and let all other supernodes in C merge with u^* by forming a BFS tree towards u^* in H. In the second case, all supernodes simply merge with the supernode in C that has smallest identifier in a similar way.

More specifically, v does the following: It first determines whether it is adjacent to an inactive supernode in G_2 by performing the Aggregate-and-Broadcast Algorithm, and, if so, informs all supernodes in $\Gamma(v)$ by sending a message to respective representatives. If no supernode receives any notification, there cannot be any inactive supernodes anymore, and u^* is simply chosen as the supernode with smallest identifier. Otherwise, we perform the Aggregate-and-Broadcast Algorithm a second time, whereby all members of v learn the supernode u^* in C that has smallest identifier among all supernodes in C that are adjacent to an inactive supernode.

If u^* is adjacent to an inactive supernode w, then u^* selects the edge $\{x, y\} \in E_2$ such that x is a member of u^* and y is a member of w such that $(\mathrm{id}(x), \mathrm{id}(y))$ is minimized using the Aggregate-and-Broadcast Algorithm. To let all other supernode select a merge edge, we first let every supernode u inform its adjacent supernodes in H about $d_u(u^*)$, which, as all nodes are unsuccessful, must be equal to $d_H(u, u^*)$. Using the Aggregate-and-Broadcast Algorithm, every supernode $v \neq u^*$ selects a merge edge to the adjacent supernode u in H that minimizes $d_u(u^*)$. Since the unsuccessful supernodes form a BFS tree in H, and merge with an inactive supernode if there exists one, we conclude the following lemma.

Lemma 3.28 (Merging Unsuccessful Supernodes)**.** *Merging unsuccessful supernodes takes time $\mathcal{O}(2^i)$, w.h.p. Propositions 3.6, 3.7, and 3.8 are maintained for the next subphase.*

3.5. Conclusion of the Algorithm

We conclude the algorithm by presenting how a well-formed tree of V as well as a spanning tree of G can be obtained from the outcome of the last subphase.

Lemma 3.29 (Outcome of the Last Subphase)**.** *At the end of the last subphase, only a single supernode v that contains all nodes of V remains. $\mathcal{E}$ forms a spanning tree of G_2.*

Proof. From Lemmas 3.27 and 3.28 we know that Proposition 3.6 holds after subphase $j = \lceil \sqrt{2^i} \rceil$ of phase $i = \lceil \log \log n \rceil - 1$. Therefore, every remaining supernode is of size

$$\min\{2^{2^{\lceil \log \log n \rceil - 1} + (\lceil \sqrt{2^{\lceil \log \log n \rceil - 1}} \rceil)^2}, n\} \geq \min\{2^{\log n/2 + \log n/2}, n\} = n,$$

which implies that there can only remain a single supernode v that contains all nodes of V. Furthermore, by Proposition 3.7 we know that $\mathcal{E}$ is a spanning tree of G_2. □

We finally construct a well-formed tree as a binary tree of height $\mathcal{O}(\log n)$ by applying the algorithm of Lemma 3.23 to $\mathcal{E}$. More precisely, we first execute the algorithm with parameter $k = \lceil \log n \rceil + 1$, for which it is guaranteed to succeed. Consider the path graph $L = (x_0, \dots, x_{n-1})$ of all nodes constructed by the algorithm (for simplicity, assume that n is a power of 2). x_0 becomes the root of our tree, and establishes $x_{2^{\log n - 1}}$ as it's only child by sending a message to it. Every node x_i that receives a message at the beginning of each subsequent round $l = \{1, \log n - 1\}$ chooses $x_{i - 2^{\log n - i - 1}}$ and $x_{i + 2^{\log n - i - 1}}$ as its children by forwarding the message to them. It is easy to see that after the last round, every node has been reached by exactly one message, which concludes the main theorem of this chapter.

Theorem 3.30 (Well-formed Tree)**.** *The algorithm constructs a well-formed tree of global edges that contains all nodes of V in time $\mathcal{O}(\log^{3/2} n)$, w.h.p.*

It remains to show how the spanning tree $\mathcal{E}$ of G_2 can be transformed into a spanning tree of the actual graph G. To that end, we again consider the path graph L constructed in the previous paragraph using the algorithm of Lemma 3.23. As a byproduct of the algorithm, each node x_i learns its index i in L. Furthermore, recall that since L contains all nodes in the order they are visited in some depth-first traversal of $\mathcal{E}$, for every $i > 0$, x_i must have at least one neighbor x_j in G_2 such that $j < i$ (i.e., x_j is visited before i in the traversal). Since each edge of G_2 either exists in G, or is the result of an introduction of two nodes with distance at most 2 in G, there exists a node x_j with $j < i$ within hop-distance 2 of x_i in G for every $i > 0$.

To obtain a spanning tree of G, in the first step every node $x_i \neq x_0$ selects an edge to a neighbor x_j in G such that $j < i$ and j is minimal. If such a neighbor does not exist, then x_i selects one of its incident edges that has not been selected by any neighbor. Such an edge must exist, because each edge of E_2 results from a path of length at most 2 in G, and thus there is a node with smaller index than i within distance 2 of x_i. It is easy to see that the selected edges do not form any cycle, since for any edge (x_i, x_j) of a cycle we either have $j < i$, or the edge is followed by an edge (x_j, x_k) such that $k < i$, leading to a contradiction. Together with the fact that only x_0 does not select an edge, we obtain the final result of this chapter.

Theorem 3.31 (Spanning Tree)**.** *After performing our algorithm, a spanning tree of G can be computed within an additional $\mathcal{O}(1)$ rounds in the local network.*

3.6. Outlook

To the best of our knowledge, there does not exist an algorithm that solves the Overlay Construction Problem in the hybrid model in time $\mathcal{O}(\log n)$ for $\gamma = \mathcal{O}(\log n)$. Our recent $\mathcal{O}(\log n)$-time solution [Göt+20], which requires polylogarithmic global capacity $\omega(\log n)$, is based on a completely different approach. Whether there exists an algorithm that solves the problem in time $\mathcal{O}(\log n)$ and with global capacity $\mathcal{O}(\log n)$ is perhaps the most intriguing question raised in this chapter.

Specifically, it is highly unclear whether one can derive a more efficient algorithm from the approach of grouping and merging supernodes. Our algorithm, for instance, does not fully exploit the power of each supernode to its limit, as we only use its *small* butterfly for communication. Furthermore, the algorithm does not take into account the initial graph topology; doing so may lead to more efficient solutions. The general approach of grouping and merging supernodes has also been used to compute minimum spanning trees. Therefore, it might be possible to extend our approach such that it computes an MST of G.

It may also be worthwhile to further investigate this problem in specific graph classes. For example, our algorithm solves the problem in time $\mathcal{O}(\log n)$ in trees, and in the same time the problem can be solved in graphs with outdegree 1 [AW07]. Other interesting classes to investigate might be planar networks or graphs with bounded arboricity.

4 Distributed Computation with Node Capacities

WHEREAS our previous discussion mainly revolved around the problem of *setting up* a suitable global network, this chapter is dedicated to studying the *communication limitations* of such a network. More precisely, we fully abstract away the issue of designing and maintaining the global network by simply assuming that the nodes form a clique. Whereas this enables each node to, in principle, contact any other node, we only allow it to send and receive a total of $\mathcal{O}(\log n)$ messages in each round. As pointed in Section 2.1, this model is an instance of the generic hybrid network model for $\lambda = 0$ and $\gamma = \mathcal{O}(\log n)$ in which each node knows the identifiers of all other nodes.

Since the model only captures the global communication aspect of hybrid networks and therefore lends itself particularly well for the study of overlay networks, we propose it as a separate model that we refer to as the *node-capacitated clique* (NCC) model. As the name implies, the model shares some similarities with the *congested clique* model by Lotker, Patt-Shamir, Pavlov, and Peleg [Lot+05], which has also been proposed as a "simple model for overlay networks" and received a lot of attention in recent years. The congested clique is an instance of the CONGEST model for the case that the graph forms a clique, and allows each node to send $\mathcal{O}(\log n)$ bits to *any* other node. In particular, it allows each node to be in contact with up to $\Theta(n)$ nodes at the same time, which makes the model much more powerful than the NCC model. For example, the gossip problem (i.e., *each* node wants to deliver one message to every other node) can be solved in a single round in the congested clique, whereas the problem requires at least $\Omega(n/\log n)$ rounds in the NCC model. Even the simple broadcast problem (i.e., *one* node wants to deliver a message to every other node) already takes time $\Omega(\log n/\log\log n)$ in our model.

Arguably, the power of the congested clique seems to severely limit its practicability. As already suggested in Section 2.1, for overlay networks in particular, the amount of communication a single node can perform typically does *not* scale with the number of nodes it can reach. Instead, it rather depends on the bandwidth of the connection of the node to the underlying communication infrastructure as a whole. Shifting the communication bounds from the graph's *edges* to its *nodes* addresses precisely this issue. We comment that the capacity bound of $\mathcal{O}(\log n)$ messages per node per round is a natural choice: It is small enough to ensure scalability, whereas any smaller bound would require unnecessarily complicated techniques for the protocol to ensure nodes do not receive more messages than the capacity bound.

Although the NCC model abstracts away the graph's topology, which is one of the main concerns of overlay network research, it is clearly closely related to overlay networks. Any distributed algorithm in which overlay edges can be established by introducing nodes to one another, and which satisfies the node capacity bound of $\mathcal{O}(\log n)$ messages, can be simulated in the NCC model without any overhead. As an example, if we are given an input graph G that has degree $\mathcal{O}(\log n)$, we can skip the sparsification step described in Section 3.2 of the previous chapter and compute a spanning tree of G in the NCC model in time $\mathcal{O}(\log^{3/2} n)$. There is also a direct relationship between the NCC and models from *parallel computing*. In fact, any algorithm for the NCC can be simulated with a multiplicative $\mathcal{O}(\log n)$ runtime overhead in the very restrictive EREW PRAM model, in which n processors can access arbitrary memory cells, but no read or write conflicts are allowed. More precisely, we assign each processor $\Theta(\log n)$ many memory cells, and simulate sending messages by letting processors write into randomly chosen cells of other processors; w.h.p., no conflicts will occur. In turn, any n-processor CRCW PRAM algorithm (which is the most permissive PRAM model that allows read conflicts and resolves write conflicts) can be simulated with a multiplicative $\mathcal{O}(\log n)$-time overhead, for example using the emulation framework of Ranade [Ran91].

In this chapter, we mainly study the computation of graph problems in the NCC. More precisely, we assume that some edges of the network are marked as edges of an undirected and connected *input graph G*. When regarding the NCC as an instance of the generic hybrid network model, this graph corresponds to the local network; however, as $\lambda = 0$, the edges do not provide any additional communication capacities. The edges of G can, for instance, be seen as edges of an underlying physical network, or represent relations between nodes in social networks.

Given an input graph G, we for example show how to compute an MST, or to solve the SSSP Problem in the node-capacitated clique. Additionally, we consider *local problems* such as the *Maximal Independent Set (MIS) Problem*, in which the goal is to identify a maximal set of nodes such that no two nodes are adjacent. Such problems are characterized by the fact that they can typically be computed *locally*, i.e., without global knowledge. However, since high-degree nodes cannot directly communicate with their neighbors in our model, even such simple problems prove to be difficult. Nonetheless, our results demonstrate that many problems can be solved efficiently with *only* global communication, indicating the power of the global network.

Underlying Publication The chapter is based on the following publication.

> J. Augustine, M. Ghaffari, R. Gmyr, K. Hinnenthal, F. Kuhn, J. Li, and C. Scheideler. "Distributed Computation in Node-Capacitated Networks". In: *Proceedings of the 31st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 69–79 [Aug+19]

We improve upon some of the results of the original publication; in particular, we show a more efficient implementation of the MST algorithm, and extend the BFS tree construction to compute exact and approximate SSSP. Furthermore, we provide an extension of our vertex coloring algorithm.

Outline In this introductory section, we review related work, summarize the contribution of this chapter, and present subsequent papers that have carried on research in the NCC. Similar to the previous chapter, we begin the technical part of this chapter with a description of algorithmic primitives in Section 4.1. The algorithms build upon the basic ideas of our already established primitives, but use much more elaborate techniques to solve more complex problems. As a first application, in Section 4.2 we show how to efficiently compute (minimum) spanning trees. The other problems considered in this chapter require the computation of an $\mathcal{O}(a)$-orientation of G, for which we present an algorithm in Section 4.3. Finally, in Section 4.4, we present a variety of algorithms to solve shortest path problems and local problems.

Related Work The congested clique model has already been studied extensively in the past years. Problems studied in prior work include routing and sorting [Len13], minimum spanning trees [JN18; GP16; Heg+15; Lot+05], subgraph detection [Cen+19b; DLP12], shortest paths [Bec+17; Cen+19a; Cen+19b; DP20], local problems [CPS20; HPS14; Gha+18], or Nash-Williams forest decompositions [GS19; BK18]. Notably, Barenboim and Khazanov [BK18] show how to solve a variety of graph problems in the congested clique efficiently given such graphs, e.g., compute an $\mathcal{O}(a)$-orientation in time $\mathcal{O}(\log a)$, an MIS in time $\mathcal{O}(\sqrt{a})$, an $\mathcal{O}(a)$-coloring in time $\mathcal{O}(a^\varepsilon)$, and an $\mathcal{O}(a^{2+\varepsilon})$-coloring in time $\mathcal{O}(\log^* n)$, where a is the arboricity of the given graph. Many other known upper bounds are astonishingly small, such as the constant-time upper bound for routing and sorting and for the computation of a minimum spanning tree, demonstrating the power of the congested clique model.

While almost no nontrivial lower bounds exist for the congested clique model (due to their connection to circuit complexity [DKO14]), various lower bounds have already been shown for the more general CONGEST model (see, e.g., [Das+12; Nan14a; KP98] and the references therein). As pointed out in [KS17], the reductions used in these lower bounds usually boil down to constructing graphs with bottlenecks, that is, graphs where large amounts of information have to be transmitted over a small cut. Applying these constructions to the NCC model is, in most cases, not directly possible. A notable exception, however, is the $\widetilde{\Omega}(n)$ lower bound for the Diameter Problem by Frischknecht et al. [FHW12]. As a careful review of the proof reveals, the lower bound construction is in fact applicable to the NCC and also implies an $\widetilde{\Omega}(n)$ lower bound here. In contrast to the obvious $\widetilde{\Omega}(n)$ lower bound for APSP in the NCC, which follows from the observation that each node has to learn $\Omega(n)$ bits, the diameter lower bound is somewhat surprising.

The graph problems considered in this chapter have already been extensively studied in many different models. In the CONGEST model, for example, a breadth-first search can trivially be performed in time $\mathcal{O}(\mathfrak{D})$. Computing weighted distances, however, is much more complicated. For the SSSP Problem, which we also study in this and the following chapters, there is a lower bound of $\widetilde{\Omega}(\sqrt{n}+\mathfrak{D})$ rounds [PR99], even for constant factor approximations [Das+12]. This bound is tight, as there is a $(1+\varepsilon)$-approximation algorithm that runs in $\widetilde{\mathcal{O}}(\sqrt{n}+\mathfrak{D})$ rounds [Bec+17]. To the best of our knowledge, the most efficient algorithms for computing exact SSSP in the CONGEST model are the ones by Ghaffari and Li [GL18] and by Forster and Nanongkai [FN18], which yield runtimes of $\widetilde{\mathcal{O}}(n^{3/4}\mathfrak{D}^{1/4})$, $\widetilde{\mathcal{O}}(\sqrt{n \cdot \mathfrak{D}})$, and $\widetilde{\mathcal{O}}(\sqrt{n}\mathfrak{D}^{1/4} + n^{3/5} + \mathfrak{D})$. In unweighted graphs, APSP and the diameter can be computed in $\mathcal{O}(n)$ rounds in the CONGEST model [PRT12; HW12], for which, as we pointed out earlier, the lower bound is tight [FHW12]. Note that this lower bound even holds for very sparse graphs [ACK16].

There exists an abundance of algorithms to solve local problems such as maximal independent set, maximal matching, and the coloring problem in the classical LOCAL and CONGEST models (see, e.g., [Bar+16] for a comprehensive overview). These, and other problems, can be characterized as problems that rely on *symmetry breaking*. More specifically, neighbors have to agree on different *roles* throughout the algorithm, e.g., being added to an MIS or not. Typically, such problems can be solved in time $\widetilde{\mathcal{O}}(1)$ both in the LOCAL and CONGEST model. As an example, in the LOCAL problem the MIS problem can be solved in time $\mathcal{O}(\log \Delta) + 2^{\mathcal{O}(\sqrt{\log\log n})}$, w.h.p. [Gha16], which comes close to the lower bound of $\Omega(\min\{\sqrt{\log n/\log\log n}, \log \Delta / \log\log \Delta\})$ that holds for the MIS and maximal matching problems [KMW04].

Whereas the running times of the above-mentioned algorithms mostly depend on n and, potentially, the degree of the graph, there have also been proposed algorithms to solve such problems more efficiently in graphs with small *arboricity* (e.g., [BEK14; BE10; BE11; Bar+16]). Typically, the algorithms make use of the Nash-Williams forest decomposition technique, which, for example, allows to compute an $\mathcal{O}(a)$-orientation in the CONGEST model in time $\mathcal{O}(\log n)$ [BE10]. As an example, the framework of Barenboim et al. [Bar+16] together with the algorithm of Ghaffari [Gha16] yields an $\mathcal{O}(\log a + \sqrt{\log n})$-time algorithm for the MIS problem.

The MST problem has also been well studied in the CONGEST model. After the seminal paper of Ghallager et al. [GHS83], who showed that an MST can be computed in time $\mathcal{O}(n \log n)$, the first worst-case optimal runtime of $\mathcal{O}(n)$ was achieved by Awerbuch [Awe87]. Although there are graphs that require $\Omega(n)$ rounds, for many graphs an MST can be computed much more efficiently, which led to a wide variety of algorithms whose runtimes are characterized by different graph properties [KP98; GKP98; Elk06]. However, due to a lower bound of $\widetilde{\Omega}(\sqrt{n})$ that holds for graphs with hop-diameter $\Omega(\log n)$ [PR99], subpolynomial runtimes can only be achieved in graphs with even smaller, e.g., constant, diameter [LPP06]. Other approaches aim at minimizing the required number of messages of an algorithm instead of optimizing its runtime. Notably, the algorithm of King et al. [KKT15] constructs an MST using time and messages $\mathcal{O}(n \log^2 n/ \log\log n)$, w.h.p., using a "graph sketching" approach. In the context of Borůvka's algorithm (see Section 2.3), this approach allows us to identify outgoing edges of components using very few messages, which leads to a very efficient MST algorithm in the NCC.

Contribution An overview of the results in this chapter can be found in Table 4.1. Note that for many important graph families such as planar graphs, our algorithms have polylogarithmic runtime (except when depending on the hop-diameter $\mathfrak{D}$ or shortest-path diameter SPD).

Although many of our algorithms rely on existing algorithms from literature, we point out that most of these algorithms cannot be executed in the node-capacitated clique in a straight-forward fashion. The main reason for that is that high-degree

Problem	Runtime	Section
Minimum Spanning Tree	$\mathcal{O}(\log^3 n / \log\log n)$	4.2
Spanning Tree	$\mathcal{O}(\log^2 n)$	4.2
BFS Tree	$\mathcal{O}((a + \mathfrak{D} + \log n)\log n)$	4.4.1
Exact SSSP	$\mathcal{O}(\min\{(a + \log n) \cdot (\mathsf{SPD} + \log n), (a + n)\log n\})$	4.4.2
$(1 + o(1))$-approx. SSSP	$\mathcal{O}((a + \mathsf{SPD}\log n)\log^2 n)$	4.4.2
Maximal Independent Set	$\mathcal{O}((a + \log n)\log n)$	4.4.3
Maximal Matching	$\mathcal{O}((a + \log n)\log n)$	4.4.4
$\mathcal{O}(a)$-Coloring	$\mathcal{O}((a + \log n)\log^{3/2} n)$	4.4.5
$\mathcal{O}(a \log n)$-Coloring	$\mathcal{O}(\log^{7/2} n)$	4.4.5

Table 4.1.: An overview of the results in this chapter. We use a for the arboricity of G, $\mathfrak{D}$ for the hop-diameter of G, and SPD for the shortest-path diameter of G (see Section 2.2). All of our results hold with high probability.

nodes cannot efficiently communicate with all of their neighbors *directly* in our model, which imposes significant difficulties to the application of the algorithms. To overcome these difficulties, we present a set of basic tools that still allow for efficient communication, and combine it with variations of well-known algorithms and novel techniques. Notably, we present an algorithm to compute an $\mathcal{O}(a)$-orientation of the input graph G with arboricity a. The algorithm is later used to efficiently construct *multicast trees* to be used for communication between nodes. Achieving this is a highly nontrivial task in our model and requires a combination of techniques, ranging from aggregation and multicasting to shared randomness and coding techniques. We believe that many of the presented ideas might also be helpful for other applications in the node-capacitated clique.

Although proving lower bounds for the presented problems seems to be difficult, we believe that many problems require a running time linear in the arboricity. For the MIS problem, for example, it seems that we need to communicate at least 1 bit of information about every edge (typically in order for a node of the edge to learn when the edge is removed from the graph because the other endpoint has joined the MIS). However, explicitly proving such a lower bound in this model seems to require more than our current techniques in proving multi-party communication complexity lower bounds.

Subsequent Work After publication of the original NCC paper, Nowicki [Now18] presented some variants and improvements of our communication primitives. Together with a random sampling technique, the author is able to solve the MST problem in time $\mathcal{O}(\log^2 n \log \Delta / \log\log n)$, which is faster than the algorithm presented in this chapter for $\Delta \ll n$, but, arguably, much more complicated. Furthermore, Nowicki describes an $\mathcal{O}(\log^2 n)$ time algorithm to compute a spanning tree. In this chapter, we offer a simple alternative to Nowicki's approach.

Augustine et al. [Aug+20a] refine the NCC model by distinguishing between the NCC_0 (see Section 2.1) and the NCC_1 model (which corresponds to the NCC). They consider $\widetilde{\mathcal{O}}(\Delta)$-time algorithms for so-called *graph realization problems*, in which the goal is to compute a topology that satisfies certain degree or connectivity properties. They assume, however, that the networks starts as a *line*, which, as we point out in Chapter 3, makes overlay construction very easy.

Finally, Robinson [Rob21] investigates the information the nodes need to learn to jointly solve graph problems and derives a lower bound for constructing spanners in the NCC. Interestingly, his result implies that spanners with constant stretch require polynomial time in the NCC, and are therefore harder to compute than MSTs.

4.1. Algorithmic Primitives

Our algorithms make heavy use of a set of communication primitives, which can be seen as extensions of the algorithms presented in Section 3.1 of Chapter 3. Similar to our previous primitives, we rely on a simulation of a butterfly network; however, in the NCC model we can simply assume that the nodes already know a butterfly network that contains all nodes, without needing to construct it explicitly. In this section, we first describe the simulation in more detail, and explain how the Aggregate-and-Broadcast Algorithm from Section 3.1 can be used to not only compute aggregates, but also achieve synchronization. We then present multiple primitives to solve more complex aggregation and dissemination problems, which will allow nodes to send and receive messages to and from specific sets of nodes associated with them. More specifically, if the number of *distinct* messages a node needs to send or receive is comparably low, even if the *total* number of messages is high, the algorithms efficiently carry out the required communication by multiplying and combining messages using the butterfly network. To do so, we use randomized routing strategies that balance the communication load among the nodes. In contrast to the Route-and-Combine Algorithm from Section 3.1, the primitives allow to efficiently communicate *arbitrarily* many messages in the network, but may require more rounds to deliver them.

Butterfly Simulation Since every node knows the identifiers of all other nodes, throughout this chapter we can assume without loss of generality that the node's identifiers are from the set $\{0, \dots, n-1\} = [n]$. In our algorithms, every node $u \in V$ with identifier $i \leq 2^d - 1$ emulates the complete column i of a d-dimensional butterfly network with $d = \lfloor \log n \rfloor$. Clearly, each node knows exactly which nodes simulate the butterfly nodes adjacent to its column. The primitives presented in this section will only require each butterfly node to send and receive at most a constant number of messages to and from adjacent butterfly nodes. Since each node simulates at most $d + 1 = \mathcal{O}(\log n)$ butterfly nodes, we can perform each communication round in the butterfly network in a single round in the NCC.

We remark that unlike the primitives introduced in Section 3.1, which solve problems defined for the butterfly network, we formally describe the upcoming problems as problems in the NCC and rely on the butterfly network simulation to solve them.

Aggregate-and-Broadcast Algorithm As a first primitive, we restate the Aggregate-and-Broadcast Algorithm from Section 3.1 in a slightly different form. Recall that as the input of the algorithm, each node of the butterfly's top level stores a data item of a multiset A, and the goal is for each butterfly node to learn $f(A)$ for some distributive aggregate function f. We generalize this problem for the NCC by assuming that each node $v \in V$ stores an arbitrary number of input values of A, and wishes to learn $f(A)$. To solve this problem, each node first combines all of its input values into a single value using the sub-aggregate function g to f. Any node v with $\mathrm{id}(v) > 2^d - 1$ sends its value to the node u such that $\mathrm{id}(u) = \mathrm{id}(v) - 2^d$. Subsequently, every node v with $\mathrm{id}(v) \leq 2^d - 1$ combines its at most two remaining values into one, and places the resulting packet at the top node of its simulated butterfly column. We can then directly apply the Aggregate-and-Broadcast Algorithm of Theorem 3.1 to let each butterfly node learn $f(A)$ in time $\mathcal{O}(\log n)$, and finally inform each node $u \in V$ with $\mathrm{id}(u) > 2^d - 1$. We correspondingly restate Theorem 3.1 for the NCC.

Theorem 4.1 (Aggregate-and-Broadcast)**.** *The Aggregate-and-Broadcast Algorithm solves any Aggregate-and-Broadcast Problem within $\mathcal{O}(\log n)$ rounds.*

In this chapter, we will also use the Aggregate-and-Broadcast Algorithm to achieve *synchronization.* Assume that the nodes of the NCC execute some distributed algorithm $\mathcal{A}$ that finishes in different rounds at the nodes. In order to start a follow-up algorithm $\mathcal{B}$ at the same round, we perform the Aggregate-and-Broadcast Algorithm in a slightly different way. Every node v with $\mathrm{id}(v) > 2^d - 1$ waits until its execution of $\mathcal{A}$ has finished, and then sends a message to node u such that $\mathrm{id}(u) = \mathrm{id}(v) - 2^d$. Any node v with $\mathrm{id}(v) \leq 2^d - 1$ also waits until $\mathcal{A}$ has finished and until it has received a message from the node u with $\mathrm{id}(u) = \mathrm{id}(v) + 2^d$, if that node exists, and then contributes a packet at the top node of its simulated column. As in the Aggregate-and-Broadcast Algorithm, all packets will be sent towards $(d, 0)$ and combined along the way. However, every butterfly node at level $i > 0$ waits until it has received packets from *both* of its neighbors at level $i - 1$ before sending a packet in the direction of node $(d, 0)$. Once node $(d, 0)$ has received packets over both of its incident edges, it knows that all nodes have finished the execution of $\mathcal{A}$. Since the subsequent broadcast will reach all butterfly nodes of the top level in the same round, we conclude the following theorem.

Theorem 4.2 (Synchronization)**.** *Assume that the nodes execute some distributed algorithm $\mathcal{A}$, and let t be the round in which the last node finishes the execution of $\mathcal{A}$. Using the Aggregate-and-Broadcast Algorithm as described above, the nodes can collectively begin a follow-up algorithm $\mathcal{B}$ in some round $t + \mathcal{O}(\log n)$.*

4.1.1. Aggregation Problem

The other algorithmic primitives are designed to solve different general communication problems, and rely on similar subroutines. For a comprehensible presentation, we arrange the problems and primitives in the order of increasing complexity, and begin with the *Aggregation Problem.* As we shall see, all other problems in this section are closely related to this problem.

Formally, in an Aggregation Problem we are given a distributive aggregate function f and a set of *aggregation groups* $\mathcal{A} = \{A_i, \dots, A_k\}$, $A_i \subseteq V$, $i \in \{1, \dots, k\}$. Each node holds exactly one *input value* $s_{u,i}$ for each aggregation group A_i of which it is a *member*, i.e., $u \in A_i$. Each aggregation group A_i has a *target* $t_i \in V$. Note that a node may be member or target of multiple aggregation groups. The goal is to aggregate all input values so that eventually t_i knows $f(s_{u,i} \mid u \in A_i)$ for all i. We define $L = \sum_{i=1}^{k} |A_i|$ to be the *global load* of the Aggregation Problem, and denote the *local load* as $\ell = \ell_1 + \ell_2$, where $\ell_1 = \max_{u \in V} |\{i \in \{1, \dots, k\} \mid u \in A_i\}|$ and $\ell_2 = \max_{u \in V} |\{i \in \{1, \dots, k\} \mid u = t_i\}|$. Whereas the global load captures the total number of messages that need to be processed, ℓ_1 and ℓ_2 indicate the work required for inserting messages into the butterfly, or sending aggregates from butterfly nodes to their targets, respectively.

We only enumerate the aggregation groups from $1, \dots, k$ to simplify the presentation of the algorithm. Actually, we only require each aggregation group to have a unique identifier. Throughout the thesis, we will slightly abuse notation and refer to an aggregation group A_i with identifier x as A_x, and to $s_{u,i}$ and t_i as $s_{u,x}$ and t_x, respectively. We assume that every node $v \in V$ knows the identifier and target of all aggregation groups it is a member of. Furthermore, v has to know an upper bound $\hat{\ell_2}$ on ℓ_2.

In the remainder of this section, we describe the *Aggregation Algorithm*, for which we prove the following theorem.

Theorem 4.3 (Aggregation)**.** *The Aggregation Algorithm solves any Aggregation Problem in time $\mathcal{O}(L/n + (\ell_1 + \hat{\ell_2})/\log n + \log n)$, w.h.p.*

The execution of the algorithm is divided into three phases, the *Preprocessing Phase*, the *Combining Phase*, and the *Postprocessing Phase*.

Preprocessing Phase In the *Preprocessing Phase*, all input values are sent in batches of size $\lceil \log n \rceil$ to butterfly nodes of level 0 chosen uniformly at random. More specifically, every node $u \in V$ transforms each input value $s_{u,i}$ for all A_i of which u is a member of into a packet $p_{u,i} = (i, s_{u,i})$. We enumerate the packets of u arbitrarily from 1 to $k \leq \ell_1$ and denote them as $p(1), \dots, p(k)$. Then, for each $j \in \{1, \dots, \lceil k/\log n \rceil\}$, u sends the packets $p((j-1)\lceil \log n \rceil + 1), \dots, p(\min\{j\lceil \log n \rceil, k\})$ in communication round j, each to a butterfly node chosen uniformly and independently at random among the butterfly nodes of level 0. To achieve synchronization after this phase, the nodes perform the Aggregate-and-Broadcast algorithm as described in Theorem 4.2.

Lemma 4.4 (Preprocessing)**.** *The Preprocessing Phase takes time $\mathcal{O}(\ell_1/\log n)$. In each round every node sends and receives at most $\mathcal{O}(\log n)$ packets, w.h.p.*

Proof. The runtime and the bound on the number of packets sent out in each round are obvious. Hence, it remains to bound the number of packets that are received in each round.

Fix any butterfly node u of level 0 and round $t \in \{1, \dots, \lceil \ell_1/\log n \rceil\}$. Altogether, at most $n\lceil \log n \rceil$ packets are sent out in round t, which we denote by $p_1, \dots, p_{n\lceil \log n \rceil}$. For each p_i, let the binary random variable X_i be 1 if and only if p_i is sent to node u in round t. Furthermore, let $X = \sum_{i=1}^{k} X_i$. We have that $\mathbb{E}[X_i] = \Pr[X_i = 1] = 1/2^d$ and, since $2^d \geq n/2$, $\mathbb{E}[X] \leq (n\lceil \log n \rceil)/2^d \leq 2\log n + 2$. Since the packets choose their destinations uniformly and independently at random, it follows from our Chernoff bound in Lemma 2.2 that $X = \mathcal{O}(\log n)$, w.h.p. □

Combining Phase The *Combining Phase* follows the idea of the Route-and-Combine Algorithm in Section 3.1, where each packet $p_{u,i}$ is routed towards a randomly chosen intermediate target t'_i, and packets that correspond to the same aggregation group are combined using the sub-aggregate function g to f. More precisely, t'_i is a node of the butterfly's bottom level chosen uniformly and independently at random using a (pseudo-)random hash function $h : \{1, \dots, k\} \to [2^d]$. However, since we allow nodes to contribute more than $\mathcal{O}(\log n)$ packets, we may not always be able to combine and forward out all packets of the same aggregation group that reach a butterfly node immediately. Instead, we need to determine an order in which packets are being sent out.

To do so, we use a variant of the *random rank protocol* [Ale82; Upf84]: Each packet $p_{u,i} = (i, s_{u,i})$ stored at some butterfly node of level 0 is assigned a *rank*, which we denote as $rank(p_{u,i}) = \rho(i)$, where $\rho : \{1, \dots, k\} \to [K]$ is a (pseudo-)random hash function known to all nodes, and $K = \widetilde{\Theta}(L/n)$ is to be determined later. All packets belonging to aggregation group A_i are routed towards their target t'_i along the unique paths on the butterfly, and using the following rules:

1. Whenever a butterfly node stores multiple packets belonging to the same aggregation group A_i, it combines them into a single packet of rank $\rho(i)$, combining their values using the sub-aggregate function g to f.
2. Whenever multiple packets from different aggregation groups contend to use the same edge in the same round, the one with smallest rank is sent (preferring the one with smallest aggregation group identifier in case of a tie), and all others get *delayed*.

Note that a packet can never get delayed by a packet belonging to the same aggregation group. Clearly, in each round at most one packet is sent along each edge of the butterfly, and eventually all (combined) packets have reached their targets.

In order to detect when the Combining Phase has finished, every node of the butterfly's top level sends out a token to all neighbors at level 1 as soon as it has sent out all of its packets. Correspondingly, every butterfly node at level $i > 0$ that has sent out all packets and has received tokens from both neighbors at level $i - 1$ sends a token to both its neighbors at level $i + 1$. By performing the Aggregate-and-Broadcast Algorithm as described in Theorem 4.2, the nodes detect when all butterfly nodes of level d have received two tokens, and agree on a round to end the Combining Phase.

Analysis of the Combining Phase To bound the runtime of the Combining Phase, we first analyze our variant of the random rank protocol in a more general setting. A path collection $\mathcal{P} = \{P_1, \dots, P_k\}$ of size k, where P_i is a directed path in a directed graph $H = (V, E_H)$, is a *leveled path collection* if every node v can be given a level $l(v) \in \mathbb{N}$ so that for every edge (v, w) of a path in that collection, $l(w) = l(v) + 1$; for example, the (directed) paths from nodes of the butterfly's top level to its bottom level form a leveled path collection. Given a leveled path collection $\mathcal{P}$ of size k, our goal is to send a packet p_i along each path P_i, where every p_i belongs to an aggregation group $A(i)$ such that packets that belong to the same aggregation group have the same destination. We define the *congestion* C of $\mathcal{P}$ as the maximum number of different aggregation groups for which there exist packets that want to cross the same edge. More formally,

$$C = \max_{e \in E_H} \{|\mathcal{P}'| \mid \mathcal{P}' \subseteq \mathcal{P}, A(i) \neq A(j)\ \forall P_i, P_j \in \mathcal{P}', e \in P_i\ \forall P_i \in \mathcal{P}'\}.$$

For the following theorem, we define the *degree* Δ of $\mathcal{P}$ as the maximum number of edges in $\mathcal{P}$ leading to the same node (i.e., the indegree of $\mathcal{P}$), and the *depth* D of $\mathcal{P}$ as the length of the longest path in $\mathcal{P}$.

Theorem 4.5 (Random Rank Combining)**.** *For any leveled path collection $\mathcal{P}$ of size k, where k is polynomial in n, that has congestion C, depth D, and degree Δ, the routing strategy used in the Combining Phase with parameter $K \geq 8C$ needs at most $\mathcal{O}(C + D\log\Delta + \log n)$ steps, w.h.p., to finish routing in $\mathcal{P}$.*

Proof. We closely follow the analysis of the random rank protocol in [Sch98] and extend it with ideas from [Lei+94] so that the analysis covers the case that packets can be combined. In order to bound the runtime, we will use the following delay sequence argument.

Suppose the runtime of the routing strategy is at least $T \geq D + s$. We want to show that it is very improbable that s is large. For this we need to find a structure that *witnesses* a large s, and which should become more and more unlikely to exist the larger s becomes.

Let p_1 be a packet that arrived at its destination v_1 in step T, and let $A(1)$ be the aggregation group of p_1. We follow the path of p_1 (or one of its predecessors, if p_1 is the result of the combination of two packets at some point) backwards until we reach an edge e_1 where it was delayed the last time. Let us denote the length of the path from v_1 to e_1 (inclusive) by l_1, and the packet that delayed p_1 by p_2; if p_1 was delayed by multiple packets, then p_2 is the last packet that delayed p_1 (i.e., it has been sent right before p_1 was sent). Let $A(2)$ be the aggregation group of p_2. From e_1 we follow the path of p_2 (or one of its predecessors) backwards until we reach an edge e_2 at which p_2 was delayed the last time, and let p_3 bet the packet from some aggregation group $A(3)$ that last delayed p_2. Let us denote the length of the path from e_1 (exclusive) to e_2 (inclusive) by l_2. We repeat this construction until we arrive at a packet p_{s+1} from some aggregation group $A(s+1)$ that prevented the packet p_s at edge e_s from moving forward, and denote the number of links on the path of p_s from e_s (inclusive) to e_{s-1} (exclusive) as l_s. Altogether, it holds for all $i \in \{1, \ldots, s\}$ that a packet p_{i+1} from aggregation group $A(i+1)$ is sent over e_i at time step $T - \sum_{j=1}^{i}(l_j + 1) + 1$, and prevents at that time step a packet p_i from aggregation group $A(i)$ from moving forward.

The path from e_s to v_1 recorded by this process in reverse order is called a *delay path.* It consists of s contiguous parts of routing paths of length $l_1, \ldots, l_s \geq 0$ with $\sum_{i=1}^{s} l_i \leq D$. Because of the contention resolution rule it holds that $\rho(i) \geq \rho(i+1)$ for all $i \in \{1, \ldots, s\}$. A structure that contains all these features is defined as follows. Note that it is deliberately defined in an abstract way to allow for an easy counting argument.

Definition 4.6 (s-delay sequence)**.** *An* s-delay sequence *consists of*

- *s not necessarily different edges $e_1, \dots, e_s$,*
- *$s+1$ not necessarily different aggregation groups $a_1, \dots, a_{s+1}$ such that there exists a packet p_i that is contained in aggregation group a_i and that traverses both e_i and e_{i-1} in that order for all $i \in \{2, \dots, s\}$, that traverses e_1 for $i = 1$, and that traverses e_s for $i = s+1$,*
- *s integers $l_1, \dots, l_s \geq 0$ such that l_1 is the number of edges on the path of p_1 from e_1 (inclusive) to its destination, and for all $i \in \{2, \dots, s\}$, l_i is the number of edges on the path of p_i from e_i (inclusive) to e_{i-1} (exclusive), and $\sum_{i=1}^{s} l_i \leq D$, and*
- *$s+1$ integers $r_1, \dots, r_{s+1}$ with $0 \leq r_{s+1} \leq \dots \leq r_1 < K$.*

An s-delay sequence is called active *if for all $i \in \{1, \dots, s+1\}$ we have $\rho(i) = r_i$.*

Our observations above yield the following lemma.

Lemma 4.7. *Any choice of the ranks that yields a routing time of $T \geq D + s$ steps implies an active s-delay sequence.*

Lemma 4.8. *The number of different s-delay sequences is at most*

$$k \cdot \Delta^D \cdot C^s \cdot \binom{D+s}{s} \cdot \binom{s+K}{s+1}.$$

Proof. There are at most $\binom{D+s}{s}$ possibilities to choose the l_i's such that $\sum_{i=1}^{s} l_i \leq D$. Furthermore, there are at most k choices for p_1, which will also fix a_1. Once v_1 and l_1 are fixed, there are at most Δ^{l_1} choices for e_1. Once e_1 is fixed, there are at most Δ^{l_2} choices for e_3, and so on. Since $\sum_{i=1}^{s} l_i \leq D$, there are at most Δ^D possibilities for $e_1, \dots, e_s$. For each edge, there are at most C different aggregation groups for which there exists a packet that traverses the edge. Therefore, there are at most C possibilities for each e_i to pick a_{i+1}, which implies that there are at most C^s possibilities to select $a_2, \dots, a_{s+1}$. Finally, there are at most $\binom{s+K}{s+1}$ ways to select the r_i such that $0 \leq r_{s+1} \leq \dots \leq r_1 < K$. $\square$

Note that we assumed that there is a unique total ordering on the ranks of the aggregation groups once ρ is fixed. Hence, every aggregation group can only occur once in an s-delay sequence. Since ρ is assumed to be a (pseudo-)random hash function, the probability that an s-delay sequence is active is $1/K^{s+1}$. Thus,

$$\begin{aligned}
& \Pr[\text{The protocol needs at least } D+s \text{ steps}] \\
\overset{\text{Lemma 4.7}}{\leq} \; & \Pr[\text{There exists an active } s\text{-delay sequence}] \\
\overset{\text{Lemma 4.8}}{\leq} \; & k \cdot \Delta^D \cdot C^s \cdot \binom{D+s}{s} \cdot \binom{s+K}{s+1} \cdot \frac{1}{K^{s+1}} \\
\leq \; & k \cdot 2^{D \log \Delta} \cdot C^s \cdot 2^{D+s} \cdot 2^{s+K} \cdot \frac{1}{K^{s+1}} \\
\leq \; & k \cdot 2^{2s + D(\log \Delta + 1) + K} \cdot \left(\frac{C}{K}\right)^s .
\end{aligned}$$

If we set $K \geq 8C$, then

$$\begin{aligned} & \Pr[\text{The protocol needs at least } D+s \text{ steps}] \\ \leq\ & k \cdot 2^{2s+D(\log\Delta+1)+K} \cdot 2^{-3s} \\ =\ & k \cdot 2^{-s+D(\log\Delta+1)+K}. \end{aligned}$$

If $k \leq n$, then Theorem 4.5 follows by setting $s = K + D(\log\Delta+1) + (\alpha+1)\log n$, where $\alpha > 0$ is an arbitrary constant. Otherwise, we choose $s = K + D(\log\Delta+1) + (\alpha+1)\log k$, and, since k is polynomial in n, obtain the result, w.h.p. □

With the help of Theorem 4.5, we are now able to bound the runtime of the Combining Phase by determining the parameters of the underlying routing problem.

Lemma 4.9 (Combining)**.** *The Combining Phase takes time $\mathcal{O}(L/n + \log n)$, w.h.p.*

Proof. The depth of the butterfly is $\mathcal{O}(\log n)$ and its degree is 4. Furthermore, the size of the routing problem is L. Therefore, it only remains to show that the congestion of the routing problem is $\mathcal{O}(L/n + \log n)$, w.h.p.

The proof is very similar to the proof of Theorem 3.2. Consider some fixed edge e from level i to $i+1$ in the butterfly. For any $A_j \in \mathcal{A}$ let the binary random variable X_j be 1 if and only if there is at least one packet from A_j that traverses e. There are $2^i \cdot 2^{d-i-1} = 2^d/2$ source-destination pairs whose unique path in the butterfly contains e, where the source is in level 0 while the destination is in level d. Since the source of every packet is chosen uniformly and independently at random among the butterfly nodes of level 0, and the destination of each aggregation group is a random butterfly node of level d, the probability for an individual packet that belongs to A_j to pass through e is $(2^d/2)/(2^d)^2 = 1/(2^{d+1})$. Hence, $\mathbb{E}[X_j] = \Pr[X_j = 1] \leq |A_j|/2^{d+1}$. Let $X = \sum_{A_j \in \mathcal{A}} X_j$. Then

$$\mathbb{E}[X] = \sum_{A_j \in \mathcal{A}} \mathbb{E}[X_j] \leq \frac{\sum_{A_j \in \mathcal{A}} |A_j|}{2^{d+1}} = \frac{L}{2^{d+1}} \leq \frac{L}{n}.$$

Since the X_j's are independent, it follows from the Chernoff bounds in Lemma 2.2 that $X = \mathcal{O}(L/n + \log n)$, w.h.p. □

Postprocessing Phase Finally, in the *Postprocessing Phase*, each intermediate target t'_i at level d sends $f(\{s_{u,i} \mid u \in A_i\})$ to the actual target t_i. To do so, every node v of level d assigns a round to each packet it needs to send, which is randomly chosen from $\{1, \ldots, \lceil \hat{\ell}_2 / \log n \rceil\}$, and sends out the packet in the assigned round. The following result can be shown similarly to Lemma 4.4, and, together with the previous lemmas, concludes the correctness of Theorem 4.3.

Lemma 4.10 (Postprocessing)**.** *The Postprocessing Phase takes time $\mathcal{O}(\hat{\ell}_2/\log n)$, w.h.p. In each round every node sends and receives at most $\mathcal{O}(\log n)$ packets, w.h.p.*

4.1.2. Multicast Tree Setup Problem

Some of our algorithms rely on a structure of precomputed *multicast trees*, which enable the nodes to multicast messages to other nodes. Formally, we define a *Multicast Tree Setup Problem* as follows. We are given a set of *multicast groups* $\mathcal{A} = \{A_1, \dots, A_k\}$, $A_i \subseteq V$, with *sources* $s_1, \dots, s_k \in V$ such that each node is source of at most one multicast group (but possibly member of multiple groups). The goal is to set up a *multicast tree* T_i in the butterfly for each $i \in \{1, \dots, k\}$ with *root* r_i, which is a node uniformly and independently chosen among the nodes of the bottom level of the butterfly, and a unique and randomly chosen *leaf* $l_{u,i}$ in the butterfly's top level for each $u \in A_i$. Let $L = \sum_{i=1}^{k} |A_i|$, $\ell = \max_{u \in V} |\{i \in \{1, \dots, k\} \mid u \in A_i\}|$ and define the *congestion* of the multicast trees to be the maximum number of trees that share the same butterfly node. We require that each node $u \in V$ knows the identifier and source of all multicast groups it is a member of. As with aggregation groups, we will sometimes refer to a multicast group A_i by a unique identifier instead of its index i.

The *Multicast Tree Setup Algorithm* solves the problem similarly to the Aggregation Algorithm; in fact, the multicast trees stem from the paths taken by the packets during an aggregation. First, every node u sends a packet $p_{u,i} = (i, \mathrm{id}(u))$ for each i such that $u \in A_i$ to a butterfly node of level 0 chosen uniformly and independently at random, which becomes the leaf $l_{u,i}$. As in the Preprocessing Phase of the Aggregation Algorithm, packets are sent in batches of size $\lceil \log n \rceil$. Then, for all i, all packets of A_i are sent towards r_i using the same routing strategy as in the Aggregation Algorithm and an arbitrary distributive aggregate function to combine packets of the same multicast group. Alongside the algorithm's execution, every butterfly node u records for every $i \in \{1, \dots, k\}$ all edges along which packets from multicast group A_i arrived during the routing towards r_i, and declares them as edges of T_i. Again, the intermediate steps are synchronized using the Aggregate-and-Broadcast Algorithm, and the final termination is determined using a token passing strategy.

The following theorem directly follows from the analysis of the Aggregation Algorithm.

Theorem 4.11 (Multicast Tree Setup)**.** *The Multicast Tree Setup Algorithm solves any Multicast Tree Setup Problem in time $\mathcal{O}(L/n + \ell/\log n + \log n)$, w.h.p. The resulting multicast trees have congestion $\mathcal{O}(L/n + \log n)$, w.h.p.*

4.1.3. Multicast Problem

Multicast trees can be used to efficiently multicast messages from a node to a set of nodes associated with it. We define such a *Multicast Problem* as follows. We assume we have constructed multicast trees for a set of multicast groups $\mathcal{A} = \{A_1, \dots, A_k\}$, $A_i \subseteq V$, with sources $s_1, \dots, s_k \in V$ such that each node is source of at most one multicast group. The goal is to let every source s_i send a message p_i to all nodes $u \in A_i$. Let C be the congestion of the multicast trees and $\ell = \max_{u \in V} |\{i \in \{1, \dots, k\} \mid u \in A_i\}|$. We require that the nodes know an upper bound $\hat{\ell}$ on ℓ.

The algorithm multicasts messages by sending them upwards the multicast trees, performing the routing strategy of the Aggregation Algorithm in "reverse order".

More precisely, every source s_i first sends p_i to the root r_i of the multicast tree T_i. Then, in the *Spreading Phase*, each r_i sends p_i to all $l_{u,i}$, $u \in A_i$. This is done by using the multicast trees and a variant of our routing protocol of the Combining Phase. First, each packet p_i is assigned a $rank(p_i) = \rho(i)$. Whenever a multicast packet p_i of some aggregation group A_i is stored by an inner node of T_i, i.e., by some butterfly node u of level $j \in \{1, \ldots, d\}$, a copy of p_i is sent over each outgoing edge of u in T_i, i.e., towards one or both of u's neighbors in level $j - 1$. If two packets from different multicast groups contend to use the same edge at the same time, the one with smallest rank is sent (preferring the one with smallest multicast group identifier in case of a tie), and the others get delayed. Once there are no packets in transit anymore, which is determined by using the same strategy as in the Aggregation Algorithm, all leaves of the multicast trees have received their multicast packet. Finally, every leaf node $l_{u,i}$ sends p_i to u in a round randomly chosen from $\{1, \ldots, \lceil \hat{\ell}/\log n \rceil\}$.

The following theorem follows from the discussion of the previous sections and an adaptation of the delay sequence argument in the proof of Theorem 4.5.

Theorem 4.12 (Multicast). *The Multicast Algorithm solves any Multicast Problem in time $\mathcal{O}(C + \hat{\ell}/\log n + \log n)$, w.h.p.*

We remark that similar to the Aggregation Algorithm, the Multicast Algorithm may be extended to allow a node to be source of multiple multicasts; however, we will only need the simplified variant in this thesis.

4.1.4. Multi-Aggregation Problem

Finally, our *Multi-Aggregation Algorithm* combines the previous primitives to allow a node to first multicast a message to a set of nodes associated with it, and then aggregate all messages destined at it. We formally define such a *Multi-Aggregation Problem* as follows. We are given a set of multicast groups $\mathcal{A} = \{A_1, \ldots, A_k\}$, $A_i \subseteq V$, with sources $s_1, \ldots, s_k \in V$ such that every source s_i stores a multicast packet p_i, and every node is source of at most one multicast group. We assume that multicast trees with congestion C have already been set up for the multicast groups. The goal is to let every node $u \in V$ receive $f(\{p_i \mid u \in A_i\})$ for a given distributive aggregate function f.

The Multi-Aggregation Algorithm essentially first performs a multicast, then maps each multicast packet to a new aggregation group corresponding to its target, and finally aggregates the packets to their targets. More precisely, first every node s_i send its multicast packet to r_i. Then, by using the same strategy as in the Multicast Algorithm, we let each $l_{u,i}$ receive p_i for all $u \in A_i$ and all i. Every node $l_{u,i}$ then *maps* p_i to a packet $(\text{id}(u), p_i)$ for all $u \in A_i$ and all i. We randomly distribute the resulting packets by letting each butterfly node of level 0 send out its packets, one after the other, to butterfly nodes of the same level chosen uniformly and independently at random. By using the same strategy as in the Aggregation Algorithm, we then aggregate all packets $(\text{id}(u), p_i)$ for all i at some intermediate target $h(\text{id}(u))$, which is chosen uniformly and independently at random among the nodes of the butterfly's bottom level using a (pseudo-)random hash function $h : \{1, \ldots, k\} \to [2^d]$. Finally, the result $f(\{p_i \mid u \in A_i\})$ is sent directly from $h(\mathrm{id}(u))$ to u, which is possible since each butterfly node stores at most $\mathcal{O}(\log n)$ results, w.h.p.

The following theorem follows from the discussion of the previous sections and from the fact that the mapping and random redistribution of packets takes time $\mathcal{O}(C)$.

Theorem 4.13 (Multi-Aggregation)**.** *The Multi-Aggregation Algorithm solves any Multi-Aggregation Problem in time $\mathcal{O}(C + \log n)$, w.h.p.*

For applications beyond the ones described in this thesis, the algorithm may also be extended to allow nodes to be source of multiple multicast groups, and to receive aggregates corresponding to distinct aggregations.

4.2. Minimum Spanning Tree

As a first example of graph algorithms for the node-capacitated clique, we describe an algorithm that computes a *minimum spanning tree* (MST) in time $\mathcal{O}(\log^3 n/\log\log n)$. More specifically, for every edge in the input graph G, one of its endpoints eventually knows whether the edge is in the MST or not. We assume that the graph G is weighted, i.e., each edge has a weight $w(e) \in \{1, 2, \dots, \mathcal{W}\} \subseteq \mathbb{N}$ as described in Section 2.1.

High-Level Description From a high level, our algorithm mimics the classic approach of Borůvka [NMN01] with Heads/Tails clustering. Similar to the algorithm of the previous chapter, the algorithm successively merges components of nodes until all nodes form a single component. Initially, each node forms a component on its own. In each of $\mathcal{O}(\log n)$ phases, every component C first flips a Heads/Tails coin with probability $1/2$. If it flips Tails, it then tries to select its lightest outgoing edge, i.e., the minimum-weight edge that leads from C to a different component. Note that we can easily make edge weights unique by concatenating them with the identifiers of its endpoints. In our algorithm, the selection of the lightest outgoing edge only succeeds with some constant probability. If C succeeds to select its lightest outgoing edge, it learns the coin flip of the component C' on the other side of the edge. If C' has flipped Heads, then the edge connecting C to C' is added to the MST, and component C *merges* with component C' (and whatever other components that are merging with C' simultaneously) into a larger component. The following observation has been used in different forms in literature (see, e.g., [KKT15; GKS17]), and we prove it for completeness.

Lemma 4.14 (Borůvka's Algorithm)**.** *Assume that each component succeeds to select its lightest outgoing edge with constant probability. After $\mathcal{O}(\log n)$ phases, all nodes form a single component, and the selected edges form a spanning tree of G.*

Proof. Since we choose the edges according to Borůvka's algorithm, at termination of our algorithm the selected edges form an MST. It remains to bound the number of phases until the algorithm terminates.

Let $1/\alpha$ be the constant probability with which a component succeeds to select its lightest outgoing edge for some $\alpha \geq 1$. Fix a current phase and, for each component

C, let X_C be the binary random variable that is 1 if C fails to select a component to merge with, i.e., it either flips Heads, or fails to select its lightest outgoing edge, or the component on the other side flips Tails (C may still successfully merge, if it is selected by a different component; however, we ignore this case). We have that $\Pr[X_C = 1] \leq 1 - (1/(4\alpha)) =: \beta$. If there are k components, then $X = \sum_C X_C$ has expected value $\mathbb{E}[X] \leq \beta k$. We call this phase *bad*, if $X \geq \varepsilon k \beta$ for some arbitrary constant $1 < \varepsilon < 1/\beta$. Otherwise, at least a $(1 - \varepsilon\beta)$-fraction of all components merge, and we call the phase *good*. By Markov's inequality, we have that

$$\Pr[\text{The phase is bad}] = \Pr[X \geq \varepsilon k \beta] \leq \mathbb{E}[X]/(\varepsilon k \beta) \leq 1/\varepsilon.$$

Since the number of components reduces by a constant factor in each good phase, we only need $\mathcal{O}(\log n)$ (say, $\rho \log n$) good rounds until all nodes form a single component.

Let $c > 0$ be a constant to be determined. Note that the probability for each round to be a good round is independent from the outcome of all previous phases. Therefore, the probability that we need more than $(1-1/\varepsilon)2\rho c \log n$ phases to terminate is bounded by the probability that of $(1 - 1/\varepsilon)2\rho c \log n$ independent Bernoulli trials with success probability at least $(1 - 1/\varepsilon)$ less than $\rho \log n$ of these trials are successful. Let X_i be the binary random variable indicating whether the i-th trial was successful, and let $X = \sum_{i=1}^{(1-1/\varepsilon)2\rho c \log n} X_i$. Clearly, $\mathbb{E}[X] \geq 2\rho c \log n$. By our second Chernoff bound in Lemma 2.2 have that

$$\Pr[X < \rho \log n] \leq \Pr[X \leq (1/2) \cdot 2\rho c \log n] \leq e^{-\rho c \log n/4} \leq n^{-c'}$$

for $\rho c \log e/4 \geq c'$, which proves the claim. □

Details of the Algorithm Over the course of the algorithm, each component $C \subseteq V$ maintains a *leader node* $l(C) \in C$ whose identifier is known to every node in the component. Furthermore, we maintain an internal multicast tree for each component C with source $l(C)$ and corresponding multicast group $C \setminus \{l(C)\}$. We will ensure that the set of multicast trees has congestion $\mathcal{O}(\log n)$, w.h.p. In each round of Boruvka's algorithm with the partition of V into components $C_1, \dots, C_N$, $C_i \subseteq V$, every leader $l(C_i)$ flips Heads/Tails and multicasts the result to all nodes in its component by using the Multicast Algorithm of Theorem 4.12. As the multicast trees have congestion $\mathcal{O}(\log n)$, and $\hat{\ell} = 1$ as every node is in exactly one component, this takes time $\mathcal{O}(\log n)$, w.h.p., by Theorem 4.12

If $l(C)$ has flipped Tails, it then learns the lightest edge to a neighbor in $V \setminus C$ in time $\mathcal{O}(\log^2 n/\log\log n)$ using the approach of King et al. [KKT15], which we describe later. Afterwards, $l(C)$ multicasts the lightest edge $\{u, v\}$, where $u \in C$ and $v \in V \setminus C$, to every node in its component, which can again be done in time $\mathcal{O}(\log n)$ using Theorem 4.12. Node u now has to learn whether v's component C' has flipped Heads, and, if so, the identifier of $l(C')$. To do so, u *joins* the multicast group $A_{\mathrm{id}(v)}$ with source v, i.e., it declares itself a member of $A_{\mathrm{id}(v)}$ and participates in the construction of multicast trees with the help of Theorem 4.11. As every node is member of at most one multicast group, setting up the corresponding trees with congestion $\mathcal{O}(\log n)$ takes time $\mathcal{O}(\log n)$, w.h.p. By using the Multicast Algorithm, the endpoints of all lightest edges learn the result of the coin flip and the identifier of their adjacent component's leader in time $\mathcal{O}(\log n)$.

If for the edge $\{u, v\}$ the component C' of v has flipped Heads, then u sends the identifier of $l(C')$, which becomes the leader of the resulting component, to its own leader, which in turn informs all nodes of C using a multicast. Note that thereby only u learns that $\{u, v\}$ is an edge of the MST, but not v. Finally, the internal multicast trees of the resulting components are rebuilt by letting each node join a multicast group corresponding to its new leader. As the components are disjoint, the resulting trees with congestion $\mathcal{O}(\log n)$ are built in time $\mathcal{O}(\log n)$, w.h.p.

Finding the Lightest Edge To find the lightest outgoing edge of a component, we "sketch" its incident edges using an implementation of the algorithm of King et al. [KKT15]. We outline the main steps of the algorithm and explain how we can execute them in our model. For the details and proof of the algorithm we refer the reader to [KKT15].

Let C be a component. The algorithm relies on the `TestOut`(j, k) procedure, which outputs for any given j and k whether there is an outgoing edge of C whose weight lies in the interval $[j, k]$. Whereas the algorithm reliably detects if *no* edge in the interval exists, it may fail to report the existence of such an edge with constant probability. To do so, every node in C learns a so-called *odd hash function* $h : \{0,1\}^{\mathcal{O}(\log n)} \to \{0,1\}$ by letting $l(C)$ multicast a random value of size $\mathcal{O}(\log n)$. The result of `TestOut`(j, k) is the value

$$\left(\sum_{v \in C} \sum_{e \in E(v,j,k)} h(\mathrm{id}(e))\right) \bmod 2,$$

where $E(v, j, k)$ is the set of incident edges of v in G whose weight lies in the interval $[j, k]$, and $\mathrm{id}(\{u, v\}) = \mathrm{id}(u) \circ \mathrm{id}(v)$ with $\mathrm{id}(u) < \mathrm{id}(v)$. Note that the hash value of any internal edge e is contributed *twice*, which effectively cancels out $h(\mathrm{id}(e))$ from the sum; therefore, only the values of outgoing edges may influence the result. Clearly, this sum can be computed in time $\mathcal{O}(\log n)$ using an aggregation in C. Note that since each subresult of the aggregation is only a single bit, we can actually perform $\mathcal{O}(\log n)$ aggregations in parallel. The authors further introduce the `HP-TestOut`(j, k) procedure, which solves the same task, but, unlike `TestOut`(j, k), is correct w.h.p. This procedure can be used to verify the result of `TestOut`(j, k), but since it relies on computing a more complex aggregate whose subresults are represented using $\mathcal{O}(\log n)$ bits, only constantly many executions of the algorithm can be performed in parallel.

King et al. present multiple applications of these procedures; in this section, we are most interested in the `FindMin-C` algorithm. Beginning with the whole range of edge weights $\{1, \ldots, \mathcal{W}\}$, the idea of the algorithm is to successively narrow down the interval using a $\lceil \log n \rceil$-ary search, each time "descending" into the interval with smallest edge weights that contains an outgoing edge. Since $\mathcal{W}$ is polynomial in n, $\mathcal{O}(\log n / \log\log n)$ descends suffice until the remaining interval contains a single edge weight, which must be the weight of the lightest outgoing edge. However, with some constant probability the algorithm may fail to narrow down the interval, wherefore $\mathcal{O}(\log n / \log\log n)$ iterations will only suffice in expectation.

Let $[j, k]$ be the interval considered in some iteration of the search. The interval is divided into $\lceil \log n \rceil$ subintervals, where, for any $i \in [\lceil \log n \rceil]$, the i-th subinterval is given by

$$\left[j + i\left\lceil\frac{(k-j+1)}{\lceil\log n\rceil}\right\rceil, j + (i+1)\left\lceil\frac{(k-j+1)}{\lceil\log n\rceil}\right\rceil\right].$$

For each subinterval $[j', k']$, `TestOut`(j', k') is invoked, where all executions are performed in parallel. Let $[j_{min}, k_{min}]$ be the i-th interval such that i is minimal and `TestOut`(j', k') returns 1, i.e., the interval with smallest edge weights for which an outgoing edge is reported. Note that the interval is guaranteed to contain an edge; however, there might still exist edges with smaller weight that the algorithm failed to detect. Therefore, the algorithm double checks whether the interval $[1, j_{min} - 1]$ does not contain an outgoing edge using `HP-TestOut`$(1, j_{min} - 1)$. If the result is 0, the interval of potential edge weights is narrowed down to $[j_{min}, k_{min}]$; otherwise, the algorithm repeats the iteration with the same interval.

The following lemma is an immediate corollary of [KKT15, Lemma 3].

Lemma 4.15 (Lightest Outgoing Edges)**.** *With constant probability, the leader node of each component learns the lightest outgoing edge of its component in within $\mathcal{O}(\log n/\log\log n)$ iterations.*

Since each iteration can be performed in time $\mathcal{O}(\log n)$, and there are $\mathcal{O}(\log n)$ phases of Borůvka's algorithm by Lemma 4.14, w.h.p., we conclude the following theorem.

Theorem 4.16 (MST)**.** *The algorithm computes a minimum spanning tree of G in time $\mathcal{O}(\log^3 n/\log\log n)$, w.h.p.*

King et al. [KKT15] also present a simpler variant of their algorithm to compute a (not necessarily minimum) spanning tree of G. The algorithm relies on the `FindAny-C` procedure, which determines *any* outgoing edge of a component C with constant probability. To do so, it only performs a constant number of aggregations and multicasts with messages of size $\mathcal{O}(\log n)$. Using similar arguments as before, it can easily be seen that the algorithm can be performed in our model in $\mathcal{O}(\log n)$ rounds. By combining the algorithm with our Head/Tails clustering approach to prevent chains of components from forming, we obtain an $\mathcal{O}(\log n)$-time algorithm to compute a spanning tree. The correctness of the following theorem follows from our previous discussion, and is very similar to the proof of [KKT15, Lemma 7].

Theorem 4.17 (Spanning Tree)**.** *The algorithm computes a spanning tree of G in time $\mathcal{O}(\log^2 n)$, w.h.p.*

4.3. Computing an $\mathcal{O}(a)$-Orientation

One of the reasons the MST problem can be solved very efficiently is because we only require *one* endpoint of each edge to learn whether the edge is in the MST or not; otherwise, the problem seems to become significantly harder, as every node would have to learn some information about each incident edge. We observe this difficulty for the other graph problems considered in this chapter as well. To approach this issue, we aim to set up multicast trees connecting each node with *all* of its neighbors in G, allowing us to essentially simulate variants of classical algorithms. As we will see, such trees can be set up efficiently if G has small arboricity by first computing an $\mathcal{O}(a)$-orientation of G, which is described in this section.

We present the *Orientation Algorithm*, which computes an $\mathcal{O}(a)$-orientation in time $\mathcal{O}((a + \log n)\log n)$, w.h.p. The orientation will allow us to set up multicast trees in time $\widetilde{\mathcal{O}}(a)$ rather than $\widetilde{\mathcal{O}}(\Delta)$, where Δ is the maximum degree of G. We construct the orientation using the Nash-Williams forest decomposition technique described in Section 2.2. Recall that the idea of this technique is to repeatedly remove nodes of remaining degree $\mathcal{O}(a)$, and direct their edges away from them, until the graph is empty. As in our case we do not know a exactly, we select the nodes to remove by comparing their remaining degree with the average degree, which still gives us an $\mathcal{O}(a)$-orientation.

More precisely, let $\deg_i(u)$ be the number of incident edges of a node u that have not yet been assigned a direction at the beginning of phase i. Define $\overline{\deg}_i$ as the average degree of all nodes u with $\deg_i(u) > 0$, i.e., $\overline{\deg}_i = \sum_{u\in V} \deg_i(u)/|\{u \in V \mid \deg_i(u) > 0\}|$. In phase i, a node u is called *inactive* if $\deg_i(u) = 0$, *active* if $\deg_i(u) \le 2\,\overline{\deg}_i$, and *waiting* if $\deg_i(u) > 2\,\overline{\deg}_i$. In each phase, an edge $\{u, v\}$ gets directed from u to v, if u is active and v is waiting, or if both nodes are active and $\mathrm{id}(u) < \mathrm{id}(v)$. Thereby, each node is waiting until it becomes active in some phase, and remains inactive for all subsequent phases. This results in a partition of the nodes into disjoint sets $V_1, \dots, V_T$, where V_i is the set of nodes that are active in phase i. We will also refer to V_i as *level* i.

Lemma 4.18 (Number of Phases)**.** *The Orientation Algorithm takes $\mathcal{O}(\log n)$ phases to compute an $\mathcal{O}(a)$-orientation.*

Proof. We show that in every phase, at least half of all nodes that are not yet inactive become inactive. Note that a node u becomes inactive in phase i if it is active in that phase, i.e., if $\deg_i(u) \le 2\,\overline{\deg}_i$. Let A_i be the set of nodes that are not inactive at the beginning of some phase i, and assume to the contrary that more than $|A_i|/2$ nodes have a degree greater than $2\,\overline{\deg}_i$. Then we arrive at a contradiction since

$$\sum_{v\in A_i} \deg_i(v) > (|A_i|/2) \cdot 2\,\overline{\deg}_i = \sum_{v\in A_i} \deg_i(v).$$

Note that any subgraph of G can be partitioned into at most a forests. Since each forest has average degree at most 2, we have that $\overline{\deg}_i \le 2a$, which implies that the computed orientation has outdegree at most $4a = \mathcal{O}(a)$. $\square$

It remains to show how a single phase can be performed efficiently in our model. Here, the main difficulty lies in having active nodes determine which of their neighbors are already inactive in order to conclude the orientations of their incident edges. We generalize this problem as an *Identification Problem*, for which we present an algorithm in the following section. Afterwards, we describe how this algorithm can be used to efficiently realize each phase of our Orientation Algorithm.

4.3.1. Identification Problem

In an *Identification Problem*, we are given a set $L \subseteq V$ of *learning* nodes and a set $P \subseteq V$ of *playing* nodes, $L \cap P = \emptyset$. Every playing node knows a subset of its neighbors that are *potentially* learning such that all neighbors that are actually learning are contained in this set. The goal is to let every learning node determine which of its neighbors are playing. In our application of this problem, the active nodes will be learning, whereas inactive nodes are playing; thereby, the active nodes will be able to identify their inactive neighbors, which enables them to assign directions to the remaining edges.

To solve any Identification Problem, we use the *Identification Algorithm*, which is described in the remainder of this section. In this section, we represent each edge $\{u, v\}$ by two directed edges (u, v) and (v, u). We assume that all nodes know s (pseudo-)random hash functions $h_1, \ldots, h_s : E \to [q]$ for some parameters s and q. The hash functions are used to map every directed edge s times randomly to q different trials. We say an edge e *participates* in trial t if there exists a $j \in \{1, \ldots, s\}$ such that $h_j(e) = t$, and denote $T_t = \{e \in E \mid \exists j \in \{1, \ldots, s\} : h_j(e) = t\}$.

Let $u \in L$. We refer to an edge (u, v) as a *red edge* of u, if v is not playing, and as a *blue edge* of u, if v is playing. Let E_u denote all outgoing edges of u, and let $R_u \subseteq E_u$ be the red edges and $B_u \subseteq E_u$ be the blue edges of u. We identify each directed edge (u, v) by the identifiers of its endpoints, i.e., $\mathrm{id}(u, v) = \mathrm{id}(u) \circ \mathrm{id}(v)$. For node u, let $\mathcal{X}(t) := \bigoplus_{e \in T_t \cap E_u} \mathrm{id}(e)$ be the XOR of the identifiers of all outgoing edges of u that participate in trial t, and $\mathcal{X}^B(t) := \bigoplus_{e \in T_t \cap B_u} \mathrm{id}(e)$ be the XOR of the identifiers of all blue edges of u that participate in trial t. Furthermore, let $x(t) := |T_t \cap E_u|$ be the total number of outgoing edges of u that participate in trial t, and let $x^B(t) := |T_t \cap B_u|$ be the number of blue edges that participate in trial t.

Obtain Required Values Our idea is to let u use these values to identify all of its red edges; from this, it can infer its blue edges, whose other endpoints must be playing. Before we describe the details of this process, we explain how the values can be computed. Clearly, the values $\mathcal{X}(t)$ and $x(t)$ can be computed by u itself for all t. The other values are more difficult to obtain as u does not know which of its edges are blue. To compute these values, we use the Aggregation Algorithm. More precisely, each playing node v is contained in aggregation group $A_{\mathrm{id}(w) \circ t}$ for every potentially learning neighbor w and every trial t such that (w, v) participates in trial t. The input of v for the aggregation group $A_{\mathrm{id}(w) \circ t}$ is $(\mathrm{id}(w, v), 1)$. The first value of the input is used to let w compute $\mathcal{X}^B(t)$, and the second value is used to compute $x^B(t)$. The aggregate function f combines any two packets that correspond to the same aggregation group by taking the XOR for the first value and the sum for the second value. Thereby, u eventually receives both $\mathcal{X}^B(t)$ and $x^B(t)$ for all $t \in [q]$.

Lemma 4.19 (Identification Runtime)**.** *Let r be an upper bound on the number of potentially learning neighbors each playing node knows. The required values can be obtained in time $\mathcal{O}(rs + q/\log n + \log n)$, w.h.p.*

Proof. In the Aggregation Problem, each playing node has to contribute rs packets, one for each aggregation group it is in. Therefore, the total number of packets is at most $L = nrs$. Furthermore, the local load ℓ_1 is at most rs. Every playing node has to receive the results of $\ell_2 = q$ aggregations, one for each of the q trials. By Theorem 4.3, this Aggregation Problem can be solved in time $\mathcal{O}(rs + q/\log n + \log n)$, w.h.p. □

Identify Red Edges We now show how u can identify its red edges using the aggregated information. First, it determines a trial t for which $x(t) = x^B(t) + 1$; if no such trial exists, the process stops. Since neighbors that are not playing did not participate in the aggregation, in this case there is exactly one red edge e such that $\mathrm{id}(e)$ is included in $\mathcal{X}(t)$ but not in $\mathcal{X}^B(t)$. Therefore, $\mathrm{id}(e)$ can be retrieved by taking the XOR of both values. Having identified $\mathrm{id}(e)$, u determines all trials j in which e participates using the collectively known hash functions and "removes" $\mathrm{id}(e)$ from $\mathcal{X}(j)$ by setting $\mathcal{X}(j) \leftarrow \mathcal{X}(j) \bigoplus \mathrm{id}(e)$ and decreasing $x(j)$ by 1. It then tries to identify another trial t for which $x(t) = x^B(t)+1$ and repeats the above process until no longer possible. Note that if u always finds a trial t for which $x(t) = x^B(t) + 1$, then it eventually has identified all red edges. Clearly, all the remaining neighbors must be playing.

Lemma 4.20 (Identification Probability)**.** *Let $u \in L$ and assume that u is incident to at most p red edges. Let s be the number of hash functions, and q be the number of trials. We have that*

$$\Pr[u \textit{ fails to identify at least } k \textit{ red edges}] < 2\left(\frac{2sk}{q}\right)^{(s-2)k/2}$$

for $q \geq 4esp$ and $s \geq 4$.

Proof. u fails to identify at least k red edges if at some iteration of the above procedure there are $j \geq k$ red edges left such that there does not exist a trial in which only one of the j edges participates. To put it differently, every remaining red edge e participates only in trials in which at least one other remaining edge participates. We denote this as event A. The j edges participate in at most $\lfloor s \cdot j/2 \rfloor$ many different trials, since otherwise there must be a trial in which only one edge participates. Therefore, the probability for event A is

$$\begin{aligned}
\Pr[A] &\leq \sum_{j=k}^{p} \binom{p}{j}\binom{q}{sj/2}\left(\frac{sj/2}{q}\right)^{sj} \\
&\leq \sum_{j=k}^{p} \left(\frac{ep}{j}\right)^{j}\left(\frac{2eq}{sj}\right)^{sj/2}\left(\frac{sj}{2q}\right)^{sj} \\
&= \sum_{j=k}^{p} \left[\left(\frac{ep}{j}\cdot\frac{sj}{2q}\right)\left(\frac{2eq}{sj}\cdot\frac{sj}{2q}\right)^{s/2}\cdot\left(\frac{sj}{2q}\right)^{s/2-1}\right]^{j} \\
&= \sum_{j=k}^{p} \left[\underbrace{\frac{e^2ps}{2q}}_{<1}\cdot\left(\frac{esj}{2q}\right)^{s/2-1}\right]^{j} \\
&< \sum_{j=k}^{p} \left(\frac{e}{2}\cdot\frac{sj}{q}\right)^{(s-2)j/2} < \sum_{j=k}^{p} \left(\frac{2sj}{q}\right)^{(s-2)j/2} \leq 2\left(\frac{2sk}{q}\right)^{(s-2)k/2},
\end{aligned}$$

where the last inequality holds because

$$\begin{aligned}
&\left(\frac{2s(j+1)}{q}\right)^{(s-2)(j+1)/2} \\
&= \left(\frac{2s(j+1)}{q}\right)^{(s-2)/2} \left(\frac{2s(j+1)}{q}\right)^{(s-2)j/2} \\
&= \left(\frac{2s(j+1)}{q}\right)^{(s-2)/2} \left(\left(\frac{j+1}{j}\right)^{j}\right)^{(s-2)/2} \left(\frac{2sj}{q}\right)^{(s-2)j/2} \\
&\le \left(\frac{2s(j+1)}{q}\right)^{(s-2)/2} e^{(s-2)/2} \left(\frac{2sj}{q}\right)^{(s-2)j/2} \\
&= \left(\frac{2es(j+1)}{q}\right)^{(s-2)/2} \left(\frac{2sj}{q}\right)^{(s-2)j/2} \\
&\le 1/2 \left(\frac{2sj}{q}\right)^{(s-2)j/2}
\end{aligned}$$

for all $j < p$. □

4.3.2. Details of the Algorithm

Finally, we show how the Identification Algorithm can be used to realize each phase of our Orientation Algorithm in time $\mathcal{O}(a + \log n)$, w.h.p. In the algorithm, every node v learns the direction of all of its incident edges in the phase in which it is active. However, every neighbor u of v that is still waiting will only learn the direction of $\{v, u\}$ in a later phase. Each phase i is divided into three *stages*: In Stage 1, every node determines whether it is active in this phase. In Stage 2, every active node learns which of its neighbors are inactive. Finally, in Stage 3 every active node learns which of its remaining neighbors, which must be either active or waiting, are active. From this information, and since every node knows the identifiers of all of its neighbors, every active node concludes the direction of each of its incident edges. In the following, we describe the three stages of phase i in detail, and prove the following lemma.

Lemma 4.21 (Single Phase)**.** *In phase i of the algorithm, every node $v \in L_i$ learns the directions of its incident edges. The phase takes time $\mathcal{O}(a + \log n)$, w.h.p.*

Stage 1: Determine Active Nodes We assume that all nodes start the stage in the same round. First, every node u that is not inactive needs to compute $\deg_i(u)$ (i.e., $\deg(u)$ subtracted by the number of inactive neighbors) to determine whether it remains waiting or becomes active in this phase. This value can easily be computed using the Aggregation Algorithm: Every inactive node v, which already knows the orientation of each of its incident edges, is a member of every aggregation group $A_{\mathrm{id}(w)}$ such that $v \to w$. As the input value of each node we choose 1, the aggregate function f is SUM, and $\ell_2 \le 1 =: \hat{\ell}_2$. By performing the Aggregation Algorithm, u determines the number of inactive neighbors, and, by subtracting the value from $\deg(u)$, computes $\deg_i(u)$. Afterwards, the nodes use the Aggregate-and-Broadcast Algorithm to compute $\overline{\deg}_i$ and to achieve synchronization. Since in the computation of $\deg_i(u)$ every inactive node is member of at most $\mathcal{O}(a)$ aggregation groups, and every active node is target of at most one aggregation group, Theorems 4.3 and 4.1 imply the following lemma.

Lemma 4.22 (Stage 1)**.** *In Stage 1, every node u learns $\deg_i(u)$ and $\overline{deg_i}$ and can infer whether it is active in phase i. Stage 1 takes time $\mathcal{O}(a + \log n)$, w.h.p.*

Stage 2: Identify Inactive Neighbors The goal of this stage is to let every active node learn which of its neighbors are inactive. The stage is divided into two *steps*: In the first step, a large fraction of active nodes *succeeds* in the identification of inactive neighbors. The purpose of the second step is to take care of the nodes that were *unsuccessful* in the first step, i.e., that only identified some, but not all, of their incident red edges. In both steps we use the Identification Algorithm described in the previous section, and carefully choose the parameters to ensure that each step only takes time $\mathcal{O}(a + \log n)$.

At the beginning of the first step, the nodes compute $\deg_i^* = \max_{u \in L_i}(\deg_i(u))$ by performing the Aggregate-and-Broadcast Algorithm. Let $\deg^* = \max_{j \le i} \deg_i^*$, which is a value known to all nodes, and note that $\deg^* = \mathcal{O}(a)$ by the proof of Lemma 4.18. Then, the nodes perform the Identification Algorithm, where the active nodes are learning and the inactive nodes are playing. Hence, the endpoints of the red edges learned by the active nodes must either be active or waiting. If we choose $s = c \log n$ and $q = 4ec \deg^* \log n$ for some sufficiently large constant $c > 4$ as parameters, then by Lemma 4.20 all nodes would learn all of their red edges, w.h.p., already in this step. However, by Lemma 4.19 with $r = \deg^*$ this would take time $\mathcal{O}(a \log n)$, since the total number of packets would be $\mathcal{O}(na \log n)$. To reduce this to $\mathcal{O}(a + \log n)$, we instead choose $s = c$ and $q = 4ec \deg^* \log n$ for some constant $c > 6$, and accept that nodes fail to identify some of their red edges in this step. More specifically, for this choice of parameters, Lemma 4.20 implies that each node fails to identify at most $\log n$ red edges, w.h.p., as we show in the following lemma.

Lemma 4.23. *In the first step, every active node fails to identify at most $\log n$ red edges, w.h.p. The execution of the Identification Algorithm takes time $\mathcal{O}(a + \log n)$, w.h.p.*

Proof. Note that every active node can only be adjacent to at most $p \le \deg^*$ active or waiting nodes, i.e., it is incident to at most p red edges. Therefore, by Lemma 4.20, the probability that an active node u fails to identify at least $\log n$ red edges is

$$2\left(\frac{2c \log n}{4ec \deg^* \log n}\right)^{(c-2)\log n/2} \le \frac{1}{2^{(c/2-1)\log n - 1}} \le \frac{1}{n^{c/2-2}}.$$

Taking the union bound over all nodes implies the first part of the lemma. The second part follows from Lemma 4.19 with parameters $r = \deg^*$, $s = c$, and $q = 4ec \deg^* \log n$. □

We now describe how these remaining edges are identified in the second step. Let $U = \{u \in V \mid u \text{ is unsuccessful}\}$, i.e., U is the set of nodes that failed to identify at least one of their red edges. We divide U into sets of *high-degree* nodes $U_{high} = \{u \in U \mid \deg(u) - \deg_i(u) > n/\log n\}$ and of *low-degree* nodes $U_{low} =$

$\{u \in U \mid \deg(u) - \deg_i(u) \leq n/\log n\}$ and consider the nodes of each set separately. Note that an unsuccessful node is a high-degree node if and only if it has more than $n/\log n$ inactive neighbors. By dealing with high-degree nodes separately, we ensure that the global load required to let low-degree nodes identify their red edges using the Identification Algorithm reduces by a $\log n$ factor compared to the execution in which *all* unsuccessful nodes participate. We make the following observation.

Lemma 4.24. *$|U_{high}| = \mathcal{O}(a + \log n)$, w.h.p.*

Proof. Let $A = \{u \in L_i \mid (\deg(u) - \deg_i(u)) > n/\log n\}$. Note that since $\overline{\deg} \leq 2a$ (see Section 2.2), we have that $\sum_{u \in V} \deg(u) \leq 2an$, and therefore $|A| \leq 2a \log n$. For $u \in A$ let X_u be the binary random variable that is 1, if u is unsuccessful in the first step, and 0, otherwise. By Lemma 4.20 and since $c > 4$, we have

$$\Pr[X_u = 1] \leq 2\left(\frac{2c}{4ec\deg^* \log n}\right)^{(c-2)/2} \leq 2\left(\frac{1}{2\log n}\right)^{c/2-1} \leq \frac{1}{\log^{c/2-1} n} \leq \frac{1}{\log n}.$$

Let $X = \sum_{u \in A} X_u$. X is the sum of independent binary random variables with expected value $\mathbb{E}[X] \leq 2a \log n/\log n = 2a =: \mu$. Let $\delta = \max\{\alpha \log n/\mu, 1\}$ for some constant $\alpha > 3$, then by using the Chernoff bound of Lemma 2.2 we get that

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\alpha \log n/3} \leq \frac{1}{n^{\alpha/3}},$$

and thus $X = \mathcal{O}(a + \log n)$, w.h.p. □

Since there cannot be very many high-degree nodes by the previous lemma, we can make their identifiers publicly known, and then let the endpoints of their incident red edges contact them directly. More specifically, we first compute $|U_{high}|$ by performing the Aggregate-and-Broadcast Algorithm. Then, each node of U_{high} sends its identifier to the node v with identifier 0 in a round that is randomly chosen from $\{1, \ldots, |U_{high}|\}$. Afterwards, v broadcasts the received identifiers, one after the other, to all nodes using the path system of the butterfly. To achieve a runtime of $\mathcal{O}(a + \log n)$, we perform the broadcasts in a *pipelined fashion*, i.e., v does not wait for the previous broadcast to terminate before starting the next broadcast.

For every node $u \in A := \{u \in V \mid u \text{ is active or waiting}\}$ we define $R_u = U_{high} \cap N(u)$, i.e., (v, u) is a red edge of v for all $v \in R_u$. Let $u \in A$. For each $v \in R_u$, u chooses a round from $\{1, \ldots, \max\{|U_{high}|, \deg_i^*\}\}$ uniformly and independently at random and sends its own identifier to v in that round.

Lemma 4.25 (Stage 2: High-Degree)**.** *Every high-degree node learns all of its red edges in time $\mathcal{O}(a + \log n)$, w.h.p.*

Proof. Since $|U_{high}| = \mathcal{O}(a + \log n)$, w.h.p., by Lemma 4.24, broadcasting all identifiers of high-degree nodes can be done in time $\mathcal{O}(a + \log n)$. By a simple application of our Chernoff bound, we can easily see that node 0 only receives $\mathcal{O}(\log n)$ messages in each round, w.h.p.

For the second part, note that $\max_{u \in A}\{|U_{high}|, \deg_i^*\} = \mathcal{O}(a + \log n)$, w.h.p., which shows that each high-degree node learns all of its red edges in time $\mathcal{O}(a + \log n)$, w.h.p. Since each active or waiting node needs to send at most $|U_{high}|$ messages, and each high-degree node receives at most $\deg_i^*$ messages, a simple application of our Chernoff bound shows each node sends and receives at most $\mathcal{O}(\log n)$ messages in each round. $\square$

To let the low-degree nodes identify their red edges, we again use the Identification Algorithm. First, in order to narrow down its set of potentially learning neighbors, every inactive node determines which of its neighbors are unsuccessful low-degree nodes. Therefore, we let every inactive node u join multicast group $A_{\mathrm{id}(v)}$ for all $u \to v$ such that v is not inactive (recall that every inactive node knows the directions of all of its incident edges, and whether the other endpoint of each edge is inactive or not). Every node $v \in U_{low}$ then informs its inactive neighbors about itself by using the Multicast Algorithm. Since every node is member of at most $\deg^*$ multicast groups, which is a value known to all nodes, the nodes know an upper bound on ℓ as required by the algorithm. Having narrowed down the set of learning nodes and the sets of potentially learning neighbors to the unsuccessful ones only, the Identification Algorithm is performed once again. As the parameters of the algorithm we choose $s = c\log n$ and $q = 4ec\log^2 n$ for some constant $c > 6$.

Lemma 4.26 (Stage 2: Low-Degree)**.** *Every low-degree node learns all of its red edges in time $\mathcal{O}(a + \log n)$, w.h.p.*

Proof. We first show that every low-degree node learns all of its red edges, w.h.p. Let $u \in U_{low}$. Since by Lemma 4.23 u has at most $p \leq \log n$ remaining red edges, by Lemma 4.20 we have that the probability that u fails to identify at most one of its remaining red edges is at most

$$2\left(\frac{2c\log n}{4ec\log^2 n}\right)^{(c\log n-2)/2} \leq \frac{1}{2^{c\log n/2-2}} \leq \frac{1}{n^{c/2-2}}.$$

Taking the union bound over all nodes implies the first part of the lemma.

For the runtime, we first consider the multicasts that are used to let inactive nodes identify which of its neighbors are low-degree nodes. By Theorem 4.11, the corresponding multicast trees are constructed in time $\mathcal{O}(a+\log n)$, as every inactive node joins at most $\deg^*$ multicast groups, and the resulting trees have congestion $\mathcal{O}(n\deg^*/n + \log n) = (a + \log n)$, w.h.p. Correspondingly, the multicast can be performed in time $\mathcal{O}(a + \log n)$, w.h.p., using Theorem 4.12.

For the final execution of the Identification Algorithm, we first prove that $\sum_{u\in U_{low}}(\deg(u) - \deg_i(u)) = \mathcal{O}(an/\log n + n)$, w.h.p., which helps us to bound the total number of packets that participate in the aggregation of Lemma 4.19 a bit more carefully. Define $A = \{u \in L_i \mid \deg(u) - \deg_i(u) \leq n/\log n\}$ (in contrast to U_{low}, A contains *all* low-degree nodes, and not only the ones that are unsuccessful in the first step of Stage 2). For a node $u \in A$, let X_u be the random variable that is $\deg(u) - \deg_i(u)$, if u is unsuccessful in the first step of Stage 2, and 0, otherwise. Then $X = \sum_{u\in A} X_u$ is a random variable that bounds $\sum_{u\in U_{low}}(\deg(u) - \deg_i(u))$. From the proof of Lemma 4.24, we have that $\Pr[X_u = \deg(u) - \deg_i(u)] \leq 1/\log n$. Since $\deg(u) - \deg_i(u) \leq \deg(u)$ for every $u \in A$, and $\sum_{u\in V}\deg(u) = 2an$, X has expected value $E[X] \leq \sum_{u \in A}(\deg(u) - \deg_i(u))/\log n \leq 2an/\log n =: \mu$. Furthermore, since $\deg(u) - \deg_i(u) \leq n/\log n$ for all $u \in A$, X is a sum of independent random variables from range $[0, n/\log n]$, we can use a Chernoff bound of Lemma 2.2 with $\delta = \max\{\alpha n/\mu, 1\}$ for some constant $\alpha > 3$, and get

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\alpha n \log n/(n3)} \leq \frac{1}{n^{\alpha/3}}.$$

Therefore, we have that $X = \mathcal{O}(an/\log n + n)$, w.h.p.

We are now able to bound the runtime of the final execution of the Identification Algorithm. Note that a straightforward application of Lemma 4.19 would yield a runtime of $\mathcal{O}(rs) = \mathcal{O}(a \log n)$, since an inactive node may still be adjacent to $\deg^*$ many unsuccessful low-degree nodes. However, we can bound the global load of the Aggregation Problem more precisely by observing that the number of edges that participate in *any* trial is bounded by $\sum_{u \in U_{low}}(\deg(u) - \deg_i(u)) = \mathcal{O}(an/\log n + n)$, w.h.p. Since each edge participates in $c \log n$ trials, the global load L can therefore be bounded by $\mathcal{O}(an + n \log n)$. Furthermore, every inactive node is a member of at most $\deg^* c \log n$ aggregation groups, and every node is a target of at most $4ec \log^2 n$ aggregation groups, which implies that $\ell_1 = \mathcal{O}(a \log n)$ and $\ell_2 = \mathcal{O}(\log^2 n)$ for the Aggregation Problem. By Theorem 4.3, the Aggregation Algorithm takes time $\mathcal{O}(a + \log n)$ to solve the problem, w.h.p. □

Stage 3: Identify Active Neighbors Finally, every active node has to learn which of the endpoints of its red edges are active. In the following, let $\mathrm{id}(e) = \mathrm{id}(u) \circ \mathrm{id}(v)$ be the identifier of an edge given by its endpoints u and v such that $\mathrm{id}(u) < \mathrm{id}(v)$. The nodes use two (pseudo-)random hash functions h and r, where h maps the identifier of an edge e to a node $h(\mathrm{id}(e)) \in V$ uniformly and independently at random, and r maps its identifier to a round $r(\mathrm{id}(e)) \in \{1, \ldots, \deg_i^*\}$ uniformly and independently at random. Every active node u sends an *edge-message* containing $\mathrm{id}(e)$ to $h(\mathrm{id}(e))$ in round $r(\mathrm{id}(e))$ for every incident edge e leading to an active or waiting node. Using this strategy, two adjacent active nodes u, v send an edge-message containing $\mathrm{id}(\{u, v\})$ to the same node in the same round. Whenever a node receives two edge-messages with the same edge identifier, it immediately responds to the corresponding nodes, which thereby learn that both endpoints are active.

Lemma 4.27 (Stage 3). *In Stage 3, every active node learns which of its neighbors are active. Stage 3 takes time $\mathcal{O}(a + \log n)$.*

Proof. By using a Chernoff bound, it can easily be shown that every active node v sends out at most $\mathcal{O}(\log n)$ edge-messages in each round. Therefore, v only receives $\mathcal{O}(\log n)$ response messages in every round. It remains to show that every node receives at most $\mathcal{O}(\log n)$ edge-messages in every round, from which it follows that it only sends out $\mathcal{O}(\log n)$ response messages in every round. Let $A = \{\{u, v\} \mid u \text{ or } v \text{ is active}\}$ and note that $|A| \leq n \deg_i^*$. Fix a node $u \in V$ and a round $i \in \{1, \ldots, \deg_i^*\}$ and let X_e be the binary random variable that is 1 if and only if $h(\mathrm{id}(e)) = u$ and $r(\mathrm{id}(e)) = i$ for $e \in A$. Then $\Pr[X_e = 1] \leq 1/(n \deg_i^*)$. $X = \sum_{e \in A} X_e$ has expected value $E[X] \leq 1$. Using our Chernoff bound, we get that $X = \mathcal{O}(\log n)$, w.h.p., which implies that u receives at most $\mathcal{O}(\log n)$ edge-messages in round i. The claim follows by taking the union bound over all nodes and rounds. □

The correctness of Lemma 4.21 follows from the previous lemmas. Together with Lemma 4.18, we obtain the following theorem.

Theorem 4.28 (Orientation Algorithm)**.** *The Orientation Algorithm computes an $\mathcal{O}(a)$-orientation in time $\mathcal{O}((a + \log n) \log n)$, w.h.p.*

4.4. Further Graph Problems

We conclude our initiating study of the node-capacitated clique by presenting a set of graph problems that can be solved efficiently in graphs with bounded arboricity. The presented algorithms rely on a structure of precomputed multicast trees. More specifically, for every node $u \in V$ we construct a multicast tree $T_{\mathrm{id}(u)}$ for the multicast group $A_{\mathrm{id}(u)} = N(u)$. Since such trees enable the nodes to send messages to all of their neighbors, in the following we refer to them as *broadcast trees*.

In a naive approach to construct these trees, one could simply use the Multicast Tree Setup Algorithm, where each node joins the multicast group of every neighbor. However, as $\ell = \Delta$, the time to construct these trees would be $\mathcal{O}(\overline{\deg} + \Delta / \log n + \log n)$, which can be $\mathcal{O}(n/\log n)$ if G is a star, for example. Instead, we first construct an $\mathcal{O}(a)$-orientation of the edges as shown in the previous section, and let u only join multicast groups $A_{\mathrm{id}(v)}$ for every out-neighbor v, which translates to injecting one packet per out-neighbor into the butterfly. Additionally, for every out-neighbor v it takes care of v joining u's multicast group. More specifically, in the application of the Multicast Tree Setup Algorithm, the packet $p_{v,\mathrm{id}(u)}$ that would be contributed by v to join u's multicast group is simply sent by u. In case of a star for example (whose arboricity is one), every node, including the center, injects at most two packets, which allows us to construct broadcast trees in time $\mathcal{O}(\overline{\deg} + \log n) = \mathcal{O}(\log n)$, w.h.p. In this example, the required orientation can be computed in time $\mathcal{O}(\log^2 n)$, w.h.p. using Theorem 4.28. In general, we obtain the following result, which follows from Theorems 4.11 and 4.28.

Lemma 4.29 (Broadcast Trees)**.** *Given an $\mathcal{O}(a)$-orientation, setting up broadcast trees takes time $\mathcal{O}(a + \log n)$, w.h.p. The congestion of the broadcast trees is $\mathcal{O}(a + \log n)$, w.h.p.*

The corollary below follows from the analysis of Theorem 4.13 and the observation that, if only *some* of the nodes for which multicast trees have been set up participate in the multi-aggregation, the runtime decreases correspondingly.

Corollary 4.30 (Neighborhood Multi-Aggregation)**.** *Let $S \subseteq V$. Using the broadcast trees, the Multi-Aggregation Algorithm solves any Multi-Aggregation Problem with multicast groups $A_{id(u)} = N(u)$ and $s_{id(u)} = u$ for all $u \in S$ in time $\mathcal{O}(\sum_{u \in S} \deg(u)/n + \log n)$, w.h.p.*

The corollary above establishes one of the key techniques used in this section. More precisely, we will exploit the fact that many distributed graph algorithms follow very simple communication patterns that can be translated into multi-aggregations in our broadcast trees.

4.4.1. Breadth-First Search Trees

As a simple example, we show how to compute BFS trees. Recall that the goal of the BFS tree problem is to let each node $v \in V$ learn its parent $\pi(v)$ in a BFS tree rooted at a given source node $s \in V$. Additionally, we want v to learn $\mathit{hop}(s,v)$. Using the broadcast trees, the problem can easily be solved by the following algorithm, which proceeds in phases. In phase 1, only s is *active*, and in phase $i > 1$, all nodes that have received an identifier in phase $i-1$ for the first time are active. In each phase, every active node sends its identifier to all of its neighbors using the broadcast trees and the Multi-Aggregation Algorithm. By choosing f as the minimum function, every node that has an active neighbor thereby receives the minimum identifier of all active neighbors. Furthermore, in every phase $i > 1$, every active node u sets $\mathit{hop}(s,u) = i-1$ and $\pi(u)$ to the node whose identifier it has received in the previous phase. Clearly, after at most $\mathfrak{D}+1$ phases, where $\mathfrak{D}$ is the hop-diameter of G, all nodes have been reached.

Theorem 4.31 (BFS Tree)**.** *The algorithm computes a BFS Tree in time $\mathcal{O}((a + \mathfrak{D} + \log n)\log n)$, w.h.p.*

Proof. The correctness of the construction of the BFS tree is obvious (see, e.g., [Pel00]). By Theorem 4.28 and Lemma 4.29, the broadcast trees are constructed in time $\mathcal{O}((a + \log n)\log n)$, w.h.p. Let S_i be the set of nodes active in phase i. By Corollary 4.30, the Multi-Aggregation Algorithm takes time $\mathcal{O}(\sum_{u \in S_i} \deg(u)/n + \log n)$ in phase i, w.h.p. Since each node is active only in one phase, we conclude a runtime of

$$\begin{aligned}
&\mathcal{O}\left((a+\log n)\log n + \sum_{i=1}^{\mathfrak{D}+1}\left(\sum_{u\in S_i} \deg^*/n + \log n\right)\right)\\
&= \mathcal{O}\left((a+\log n)\log n + \left(\sum_{u \in V} \deg(u)/n\right) + (\mathfrak{D}+1)\log n\right)\\
&= \mathcal{O}((a+\mathfrak{D}+\log n)\log n),\ \text{w.h.p.}
\end{aligned}$$

□

4.4.2. Single-Source Shortest Paths

BFS trees, which contain *unweighted* shortest paths, can be computed very efficiently using our techniques because every node effectively only has to send a message to its neighbors *once*. As already pointed out in Section 2.3, computing *weighted* distances to s is more challenging, since a path between u and v that has fewest hops may not be a shortest path. In this section, we present some algorithms for the SSSP Problem, in which each node u wants to learn $d(s,u)$ for a given source node s. Note that a shortest-path tree, which is the weighted counterpart of a BFS tree, can easily be obtained afterwards by simply determining the neighbor v of each node u that minimizes $d(s,v) + w(\{v,u\})$ using a multi-aggregation, which, given an $\mathcal{O}(a)$-orientation, can be done in an additional $\mathcal{O}(a + \log n)$ rounds.

Exact SSSP Our first algorithm for the SSSP Problem is a simple adaptation of the distributed Bellman-Ford algorithm. From a high level, every node $v \in V$ starts with a distance value $\tilde{d}(s,v)$, where $\tilde{d}(s,s)=0$, and $\tilde{d}(s,v)=\infty$ for all $v \in V \setminus \{s\}$. In each iteration, every node $v \in V$ sends $\tilde{d}(s,v)+w(\{v,u\})$ to every node $u \in N(v)$. Then, u updates $\tilde{d}(s,v)$ to $\min\{\tilde{d}(s,v), \min A\}$, where A is the set of values received by u in this iteration. After $\mathsf{SPD}+1$ iterations, where SPD is the shortest-path diameter of G, we have that $\tilde{d}(s,v)=d(s,v)$ for every node $v \in V$.

The algorithm lends itself for an implementation in the NCC using our techniques. First, we again compute an $\mathcal{O}(a)$-orientation and construct broadcast trees in time $\mathcal{O}((a+\log n)\log n)$, w.h.p., using Theorem 4.28 and Lemma 4.29. Each round of Bellman-Ford translates into a multi-aggregation in which all nodes contribute their current distance value. However, since a distance value sent from v to u needs to be augmented by the weight $w(\{v,u\})$ before being aggregated towards u, the leaf node $l_{u,\mathrm{id}(v)}$ of v's broadcast tree needs to know that weight. Clearly, we can easily ensure this when constructing the broadcast trees. After receiving the distance value $\tilde{d}(s,v)$ from v during the multi-aggregation, $l_{u,\mathrm{id}(v)}$ can contribute $\tilde{d}(s,v)+w(\{v,u\})$ in the aggregation towards u. By Corollary 4.30, the multi-aggregation takes time $\mathcal{O}(a+\log n)$, w.h.p.

After each multi-aggregation, we use the Aggregate-and-Broadcast Algorithm to check whether the distance value of some node changed; once this is not true anymore, the algorithm terminates. Since the distance values cease to change exactly when all nodes have obtained their true distance, we conclude the following theorem.

Theorem 4.32 (Bellman-Ford)**.** *Using an adaptation of the distributed Bellman-Ford algorithm, SSSP can be solved exactly in time $\mathcal{O}((a+\log n)\cdot(\mathsf{SPD}+\log n))$, w.h.p.*

Note that by limiting the number of iterations of the algorithm to $\mathcal{O}(h)$, we can also solve h-limited SSSP in time $\mathcal{O}((a+\log n)\cdot(h+\log n))$, w.h.p. Furthermore, (h,k)-SSP can obviously be solved in time $\mathcal{O}((a+\log n)\cdot(hk+\log n))$ by sequentially performing k executions.

The downside of Theorem 4.32 is the fact that the runtime may be up to $\widetilde{\Theta}(n^2)$ if both the arboricity and the shortest-path diameter of G are high. Here, it may be more efficient to conduct a “sequential” execution of Dijkstra's algorithm. In Dijkstra's algorithm, each node $v \in V$ initializes its distance value $\tilde{d}(s,v)$ as in the previous algorithm. However, in each iteration we only pick the node that currently has smallest distance value, and let this node send messages to their neighbors, which in turn update their distance value. It is well-known that each node is only picked once until $\tilde{d}(s,v)=d(s,v)$ for every $v \in V$, which leads to the following theorem.

Theorem 4.33 (Dijkstra)**.** *Using an adaption of Dijkstra's algorithm, SSSP can be solved exactly in time $\mathcal{O}((a+n)\log n)$, w.h.p.*

Proof. In each iteration, we pick a different node using the Aggregate-and-Broadcast Algorithm and then perform a multi-aggregation. After n iterations, all nodes have been considered. From Theorem 4.28, Lemma 4.29, and Corollary 4.30 we conclude a runtime of

$$\mathcal{O}\left((a+\log n)\log n + \sum_{v\in V}(\deg(v)/n+\log n)\right) = \mathcal{O}((a+n)\log n), \text{ w.h.p.} \qquad \square$$

By performing both algorithms for exact SSSP in parallel, and determining which terminates first using the Aggregate-and-Broadcast Algorithm, we can summarize a runtime of $\mathcal{O}(\min\{(a+\log n)\cdot(\mathsf{SPD}+\log n),(a+n)\log n\})$, w.h.p.

Approximate SSSP For some values of a and SPD we can derive more efficient algorithms by considering *approximate* solutions, as we show in this section. We use the approach of Nanongkai [Nan14a], whose idea is to reduce the problem to $\mathcal{O}(\log n)$ instances of the unweighted SSSP Problem, which, as we have argued above, requires much less communication. The approach is based on the following theorem, which is proven in [Nan14b].

Theorem 4.34 ([Nan14b, Theorem 3.3])**.** *Let $h \in \mathbb{N}$ and $\varepsilon = 1/\log n$. For any $i \in \mathbb{N}$ and edge $\{x,y\} \in E$, let $w^i(\{x,y\}) = \lceil \frac{2hw(\{x,y\})}{\varepsilon 2^i} \rceil$. For any nodes $u, v \in V$, let*

$$\tilde{d}_h(u,v) = \min\left\{\frac{d^i(u,v)\cdot\varepsilon 2^i}{2h} \mid i \in N : d^i(u,v) \le (1+2/\varepsilon)h\right\},$$

where $d^i(u,v)$ is the distance between u and v in the graph G^i in which the weight of each edge $\{x,y\}$ is changed to $w^i(\{x,y\})$. Note that $\tilde{d}_h(u,v) = \infty$ if the above set is empty. We have that

$$d_h(u,v) \le \tilde{d}_h(u,v) \le (1+\varepsilon)d_h(u,v),$$

where $d_h(u,v)$ is the h-limited distance from u to v as defined in Section 2.2.

Note that for any given i and h, every node $v \in V$ knows $w^i(\{v,u\})$ for all $u \in N(v)$. To evaluate the expression of $\tilde{d}_h(u,v)$ for a given u, v needs to learn $d^i(u,v)$ for all $i \in \mathbb{N}$. However, for sufficiently large $i = \mathcal{O}(\log n)$, the weights of all edges reduce to 1; therefore, we actually only need to compute $d^i(u,v)$ for $\mathcal{O}(\log n)$ many different i.

Therefore, to solve $(1+o(1))$-approximate SSSP, each node $v \in V$ needs to compute $d^i(s,v)$ for $\mathcal{O}(\log n)$ different i. As Nanongkai points out, for any given i the values can be computed by using a simple generalization of the BFS algorithm [Nan14b, Algorithm 3.2]. The algorithm can be stated as follows. Initially, every node sets $\tilde{d}^i(u,v) = \infty$, and the source node s sends a message with distance value 0 to itself. In iteration $j = 1, \dots, (1+2/\varepsilon)h+1$, every node updates $\tilde{d}^i(s,v)$ to $\min\{\tilde{d}^i(s,v), \min\{\ell \in A_v \mid \ell \le (1+2/\varepsilon)h\}\}$, where A_v is the set of distance values received so far by v. Furthermore, if $j = \tilde{d}^i(s,v)+1$, then v sends a message that contains the distance value $\tilde{d}^i(u,v) + w^i(\{v,u\})$ to each node $u \in N(v)$. After the last iteration, $\tilde{d}^i(s,v) = d^i(s,v)$ if $d^i(s,v) \le (1+2/\varepsilon)h$; otherwise, $\tilde{d}^i(s,v) = \infty$. Furthermore, as Nanongkai points out, every node sends out messages in only *one* iteration. Therefore, by using multi-aggregations with existing broadcast trees, an execution of the algorithm for a given i can be performed in time $\mathcal{O}(a\log n + (1+2/\varepsilon)h\log n) = \mathcal{O}((a+h\log n)\log n)$. Since setting up the broadcast trees takes time $\mathcal{O}((a+\log n)\log n)$, and we need to perform $\mathcal{O}(\log n)$ executions, we have the following theorem.

Theorem 4.35 (Approximate h-hop SSSP)**.** *The algorithm solves $(1+o(1))$-approximate h-hop SSSP in time $\mathcal{O}((a+h\log n)\log^2 n)$, w.h.p.*

For $h = \mathsf{SPD}$ we have the following corollary. Note that the algorithm is much more efficient than our exact solutions for example if $a = \mathsf{SPD} = \Theta(\sqrt{n})$.

Corollary 4.36 (Approximate SSSP)**.** *The algorithm solves $(1 + o(1))$-approximate SSSP in time $\mathcal{O}((a + \mathsf{SPD} \log n) \log^2 n)$, w.h.p.*

We believe that the above results can also be extended to also approximate (h, k)-SSP in time $\widetilde{\mathcal{O}}(h + k)$ by starting the executions from each of the k sources with some random delay [Nan14a]. Together with the *randomized hitting set construction* [UY91], which has been very popular in recent years (see, e.g., [Aug+20b; Nan14a; FN18]), this technique should allow us to approximate SSSP in time $\widetilde{\mathcal{O}}(a + \sqrt{n})$.

4.4.3. Maximal Independent Set

In this section, we show how to compute a *maximal independent set (MIS)*: A set $U \subseteq V$ is an MIS if (1) it is an independent set, i.e., no two nodes of U are adjacent in G, and (2) there is no independent set $U' \subseteq V$ such that $U \subset U'$. The first distributed algorithms to solve this classical graph problem go back to Luby [Lub86] and Alon et al. [ABI86]. We use the algorithm of Métivier et al. [Mét+11], which lends itself for an implementation in our model, and which works as follows. First, all nodes are active and no node is in the MIS. The algorithm proceeds in phases, where in each phase every active node u first chooses a random number $r(u) \in [0, 1]$ and broadcasts the value to all of its neighbors. u then joins the MIS (and becomes inactive) if $r(u)$ is smaller than the minimum of all received values. If so, it broadcasts a message to all of its neighbors, instructing them to become inactive.

We can easily perform each phase of the algorithm in our model by using two multi-aggregations, the first to let every node aggregate the minimum of all values chosen by its neighbors, and the second to let every node that is not in the MIS determine whether it is adjacent to a node that is in the MIS. This information is then used to determine whether the nodes have reached an MIS using the Aggregate-and-Broadcast Algorithm. Since $\mathcal{O}(\log n)$ phases suffice, w.h.p. [Mét+11], and each phase can be performed in time $\mathcal{O}(a + \log n)$ by Corollary 4.30, we conclude the following theorem.

Theorem 4.37 (MIS)**.** *The algorithm computes a maximal independent set in time $\mathcal{O}((a + \log n) \log n)$, w.h.p.*

4.4.4. Maximal Matching

Similar to an MIS, a *maximal matching $M \subseteq E$* is defined as a maximal set of independent (i.e., node-disjoint) edges. Therefore, a maximal matching of G corresponds to an MIS in the *line graph $L(G)$*, which contains a node for every edge of G and an edge $\{u, v\}$ if the corresponding edges of u and v are incident in G. However, computing a maximal matching by simulating the MIS algorithm in $L(G)$ is quite technical. Instead, we propose to use the algorithm of Israeli and Itai [II86], which works as follows. Initially, no node is matched. The algorithm proceeds in phases, where in each phase every unmatched node u performs the following procedure. First, it chooses an edge to an unmatched neighbor uniformly at random. If

u itself has been chosen by multiple neighbors, it accepts only one choice arbitrarily and informs the respective node. The outcome is a collection of paths and cycles. Each node of a path or cycle finally chooses one of its at most two neighbors. If thereby two adjacent nodes choose the same edge, the edge joins the matching and the two nodes become matched. Afterwards, all matched nodes and their incident edges are removed from the graph.

The algorithm can easily be realized using our communication primitives. First, we let every unmatched node randomly pick one of its unmatched neighbors by performing the Multi-Aggregation Algorithm with a slight modification. Here, every node u that is still unmatched multicasts a packet $p_{\mathrm{id}(u)}$ using its broadcast tree. Recall that after $p_{\mathrm{id}(u)}$ has reached butterfly node $l_{v,\mathrm{id}(u)}$ for all $v \in N(u)$ in the execution of the Multi-Aggregation Algorithm, it is mapped to a new packet $(\mathrm{id}(v), p_{\mathrm{id}(u)})$. Here, we additionally let $l_{v,\mathrm{id}(u)}$ choose a value $r \in [0,1]$ uniformly at random, and annotate $(\mathrm{id}(v), p_{\mathrm{id}(u)})$ by r. Whenever thereafter two packets with the same target are combined, the packet annotated by the minimum value remains. Thereby, every node that still has an unmatched neighbor receives the identifier of a node chosen uniformly and independently at random among its unmatched neighbors.

Afterwards, every node that has been chosen by multiple neighbors has to choose one of them arbitrarily. This can be done by performing the Aggregation Algorithm, in which we let every node u aggregate the minimum of the identifiers of all nodes by which it has been chosen in the previous step. In the resulting collection of paths and cycles, neighbors can directly send messages to each other to determine which edges join the matching. Finally, the nodes have to determine whether the matching is maximal, which can be done as described in the previous section. Using Corollary 3.5 of [II86] and Chernoff bounds, it can be shown that $\mathcal{O}(\log n)$ phases suffice. Since each phase takes time $\mathcal{O}(a + \log n)$ by Corollary 4.30, we conclude the following theorem.

Theorem 4.38 (Maximal Matching)**.** *The algorithm computes a maximal matching in time $\mathcal{O}((a + \log n) \log n)$, w.h.p.*

4.4.5. $\mathcal{O}(a)$-Coloring

The goal of this section is to compute an $\mathcal{O}(a)$-*coloring*, in which every node has to choose one of $\mathcal{O}(a)$ colors such that no color is chosen by two adjacent nodes. Following the idea of Barenboim and Elkin [BE10], we consider the partition of nodes into levels $V_1, \dots, T_T$ and color the nodes of each level separately. Recall that after the execution of the Orientation Algorithm in Section 4.3, every node knows the index of its own level. Furthermore, for all i every node $u \in V_i$ knows which of its neighbors are in *lower levels* $V_1, \dots, V_{i-1}$, the same level V_i, and *higher levels* $V_{i+1}, \dots, V_T$, since it knows which of its neighbors were inactive, active, or waiting in phase i. First, the nodes use the Aggregate-and-Broadcast Algorithm to compute $\hat{a} = \max_{u \in V}\{\max\{\deg_L(u), \deg_{\mathrm{out}}(u)\}\} = \mathcal{O}(a)$, where $\deg_L(u)$ is the number of neighbors of u that are in the same level as u and $\deg_{\mathrm{out}}(u)$ is the outdegree of u in the computed orientation. Note that $\hat{a} \leq 4a$, since each node has at most $4a$ neighbors in the same or in higher levels (see Lemma 4.18). Furthermore, the nodes set up multicast trees for multicast groups $A_{\mathrm{id}(u)} = \mathrm{N}_{\mathrm{in}}(u)$ with source $s_{\mathrm{id}(u)} = u$ for all $u \in V$, where $\mathrm{N}_{\mathrm{in}}(u)$ is the set of u's in-neighbors. To do so, every node joins the multicast group of each of its out-neighbors, which, given an $\mathcal{O}(a)$-orientation, can be done in time $\mathcal{O}(a + \log n)$, w.h.p., by Theorem 4.11.

Afterwards, the algorithm proceeds in phases $1, \dots, T$, where in each phase i the nodes of level V_{T-i+1} get colored. Throughout the algorithm's execution, every node u maintains a color palette $C(u)$ that is initially set to $[(2+\varepsilon)\hat{a}] \subseteq [4(1+\varepsilon)a]$ for some constant $\varepsilon > 0$. After each phase, the color palette of every remaining uncolored node has been narrowed down to all colors that have not yet been chosen by its neighbors. Since every $u \in V_{T-i+1}$ has at most $\hat{a}$ neighbors in higher levels, $C(u)$ still consists of at least $(1+\varepsilon)\hat{a}$ colors at the beginning of phase i.

In phase i of the algorithm, the nodes of level V_{T-i+1} essentially perform the *Color-Random Algorithm* of Kothapalli et al. [Kot+06]. Note that the algorithm requires that there are no short directed cycles among the participating nodes. Since edges between nodes of the same level lead from lower to higher identifier, which implies that there cannot be *any* cycles in the same level, we can apply the algorithm to each level. The degree within each level is at most $\hat{a}$, and each node starts with a color palette that contains at least $(1+\varepsilon)\hat{a}$ colors. Therefore, [Kot+06, Theorem 4.4] implies that at termination of the algorithm each node has picked a color, w.h.p.

It remains to describe how the algorithm works and how long it takes until each node of the level has chosen a color. First, every node $u \in V_{T-i+1}$ chooses a color c_u from its remaining color palette uniformly at random. Then, it informs its in-neighbors about its choice by performing the Multicast Algorithm using the precomputed multicast trees and $\hat{a}$ as an upper bound on ℓ. Thereby, u receives the colors chosen by its out-neighbors of the same level. If u does not receive its own color c_u, it permanently chooses c_u. In that case, it first informs all of its in-neighbors about its permanent choice by again performing the Multicast Algorithm. Afterwards, it informs all of its out-neighbors by performing the Aggregation Algorithm. Here, u is a member of aggregation groups $A_{\mathrm{id}(v) \circ c_u}$ for all $v \in \mathrm{N}_{\mathrm{out}}$ and target of aggregation groups $A_{\mathrm{id}(u) \circ i}$ for all $i \in [(2+\varepsilon)\hat{a}]$. Note that every node is a member of at most $\hat{a}$ and a target of at most $(2+\varepsilon)\hat{a}$ aggregation groups. Afterwards, all nodes (including nodes of lower levels) remove all colors permanently chosen by neighbors from their color palettes.

The above procedure is repeated until all nodes of level V_{T-i+1} have permanently chosen a color, which is determined by performing the Aggregate-and-Broadcast Algorithm after each repetition. By [Kot+06, Lemma 4.2], after $\mathcal{O}(\sqrt{\log n})$ repetitions any simple oriented path of length $\sqrt{\log n}$ will have at least one colored node, w.h.p. Therefore, every component of uncolored nodes within V_{T-i+1} is a directed acyclic graph with depth at most $\sqrt{\log n}$, which implies that the nodes of each such component will be colored after an additional $\mathcal{O}(\sqrt{\log n})$ repetitions (see [Kot+06, Lemma 4.3]). We have the following theorem.

Theorem 4.39 (Arboricity Coloring)**.** *The algorithm computes an $\mathcal{O}(a)$-coloring in time $\mathcal{O}((a + \log n)\log^{3/2} n)$, w.h.p.*

Proof. Clearly, there are $\mathcal{O}(\log n)$ phases. By the analysis of the algorithm of Kothapalli et al. [Kot+06], each phase requires $\mathcal{O}(\sqrt{\log n})$ repetitions until all nodes are colored. To realize each phase, we need broadcast trees that can be set up in time

$\mathcal{O}((a+\log n)\log n)$, w.h.p., by Theorem 4.28 and Lemma 4.29. Since each multicast and aggregation can be performed in time $\mathcal{O}(a+\log n)$, w.h.p., each repetition of a phase takes time $\mathcal{O}(a+\log n)$, w.h.p. $\square$

If $a = \Omega(\log n)$, then the algorithm can be improved to compute an $\widetilde{\mathcal{O}}(a)$-coloring in time $\widetilde{\mathcal{O}}(1)$ by randomly partitioning the graph into low-arboricity components (see, e.g.,[GS19]). The approach can be summarized in the following lemma.

Lemma 4.40 (Low-Arboricity Partition)**.** *Let $a^* \geq a(G)$ be an upper bound on the arboricity of G with $a^* = \mathcal{O}(a(G))$. Assume that each node $v \in V$ picks a value $c(v) \in [a^*]$ independently and uniformly at random. Then the arboricity of the subgraph G_i induced by edges $\{u, v\}$ such that $c(u) = c(v) = i$ is at most $a(G_i) = \mathcal{O}(\log n)$, w.h.p.*

Proof. Since a^* is an upper bound on the arboricity of G, there exists an orientation of G with outdegree a^*. Fix such an orientation, and consider some node $v \in V$. Let $v_1, \dots, v_{a^*}$ be the out-neighbors of v, and let X_i be the random variable that indicates whether $c(v_i) = c(v)$. Clearly, $\Pr[X_i = 1] = 1/a^*$, and $X = \sum_{i=1}^{a^*}$ is a sum of independent binary random variables with expected value $E[X] \leq 1 =: \mu$. By our Chernoff bound, at most $\mathcal{O}(\log n)$ out-neighbors of v will choose the same value as v. By taking the union bound over all nodes, we have that each node has at most $\mathcal{O}(\log n)$ out-neighbors with the same value. Therefore, any component of G_i admits an $\mathcal{O}(\log n)$-orientation, and G_i has arboricity $\mathcal{O}(\log n)$, w.h.p. $\square$

Therefore, if the nodes know an upper bound on a, they can partition the graph accordingly, using the fact that given shared randomness each node can infer which of its neighbors picked the same value. Then, we can construct an $\mathcal{O}(\log n)$-orientation using Theorem 4.28, and establish broadcast trees that connect each node with its neighbors in the same component using Lemma 4.29 in time $\mathcal{O}((a+\log n)\log n) = \mathcal{O}(\log^2 n)$, w.h.p. The algorithm of Theorem 4.39 computes an $\mathcal{O}(\log n)$-coloring in each component in time $\mathcal{O}(\log^{5/2} n)$, and, by using a different color palette for each G_i, we can obtain an $\mathcal{O}(a \log n)$-coloring for G.

If a is not known, we could of course compute an upper bound by using the Orientation Algorithm; however, this would require time $\widetilde{\mathcal{O}}(a)$. Instead, we simply *guess* a by starting with $a^* = 1$, and repeat the above process until we actually obtain an $\mathcal{O}(\log n)$-orientation in each component, doubling the value a^* after each unsuccessful repetition. Clearly, this only adds an $\mathcal{O}(\log n)$ factor to the overall runtime, leading to the following theorem.

Theorem 4.41 (Polylog Coloring)**.** *The algorithm computes an $\mathcal{O}(a \log n)$-coloring in time $\mathcal{O}(\log^{7/2} n)$, w.h.p.*

4.5. Outlook

This chapter studies the effect of node-capacities on the complexity of distributed graph computations. Our ideas to approach the difficulties such limitations impose might be of interest for other problems as well. Clearly, there is an abundance of classical problems that may be newly investigated under the NCC model and for which our algorithms may be helpful. In general, it would be interesting to see a classification of graph algorithms that can or cannot be efficiently performed in this model. We are also very interested in proving lower bounds, which seems to be highly nontrivial. Particularly, we do not know whether the arboricity or the average node degree are natural lower bounds for some of the problems considered in this chapter, although we highly suspect it.

Interestingly, the algorithms presented in this chapter do not fully exploit the power of the node-capacitated clique. In fact, *all* algorithms we presented can be performed only with the help of some suitable overlay such as a well-formed tree. The capability to communicate with all nodes in the network directly is not actually required. Since, however, the NCC is a very simple and clean model for global communication, we will use it as the global network model in the upcoming chapters of this thesis, but never exploit its full power. Beyond the results in this thesis, it might be interesting to investigate problems for which this power is actually necessary.

5 Shortest Paths in Sparse Hybrid Networks

SHORTEST path problems are among the most famous and well-studied problems in distributed computing. They are relevant for a variety of practical applications such as finding short routing paths, determining distances and delays, or learning properties of the topology of a network. For hybrid networks, such insights may be important to select neighbors to forward packets to, or to choose the appropriate communication mode to perform tasks most efficiently. From a theoretical perspective, shortest path problems are particularly interesting because they capture global graph properties that are usually very hard to compute. As the related work section of the previous chapter indicates, there is an abundance of research that revolves around efficiently computing shortest path problems in distributed systems.

In this chapter, we focus on the Single-Source Shortest Paths (SSSP) and the Diameter Problem (see Section 2.3 for a formal definition). In contrast to the APSP Problem, for which there is a lower bound of $\widetilde{\Omega}(\sqrt{n})$ [Aug+20b] that even holds in the most generous LOCAL+NCC model, in many cases these problems can be computed very efficiently. More specifically, we consider the CONGEST+NCC model, in which only $\lambda = \mathcal{O}(1)$ messages can be sent over each local edge, and only $\gamma = \mathcal{O}(\log n)$ messages can be communicated via global edges at each node in each round. Thereby, we only grant the nodes very limited communication capabilities for both communication modes, disallowing them, for example, to gather complete neighborhood information to support their computation.

In the CONGEST+NCC model, however, computing SSSP and the diameter is still difficult if the graph G under consideration is very *dense*. As an example, the Diameter Problem takes time $\widetilde{\Omega}(n^{1/3})$ in general graphs (even with unbounded local communication) [KS20]. This motivates us to study these problems in sparse graphs. Specifically, we present randomized exact $\mathcal{O}(\log n)$ time algorithms for the SSSP and Diameter Problem in *cactus graphs*. Formally, a connected graph G is a cactus graph if any two cycles share at most one node. Cactus graphs are relevant for wireless communication networks, where they can model combinations of star/tree and ring networks [Ben+12], or combinations of ring and bus structures in LANs [LW00]. However, research on solving graph problems in cactus graphs mostly focuses on the sequential setting (e.g., [Ben+12; LW00; BH17; LWS99]). Furthermore, we present 3-approximate randomized algorithms for SSSP and the Diameter Problem with runtime $\mathcal{O}(\log^2 n)$ for graphs that contain at most $n + \mathcal{O}(n^{1/3})$ edges and have arboricity $\mathcal{O}(\log n)$. Our algorithms for sparse graphs are exponentially faster than the best known algorithms for general graphs [Aug+20b; KS20; CLP20], which actually require the much more powerful LOCAL+NCC model.

Underlying Publication The chapter is based on the following publication.

> M. Feldmann, K. Hinnenthal, and C. Scheideler. "Fast Hybrid Network Algorithms for Shortest Paths in Sparse Graphs". In: *Proceedings of the 24th International Conference on Principles of Distributed Systems (OPODIS).* 2020, 31:1–31:16 [FHS20]

Related and Subsequent Work To the best of our knowledge, shortest path problems in our hybrid network model have only been considered for the LOCAL+NCC variant (for a more comprehensive summary, see the related work section in Chapter 6). The initial publication of Augustine et al. [Aug+20b], upon which Chapter 6 is based, considers the APSP and SSSP Problem and presents both exact and approximate solutions. Notably, the publication shows that even $\widetilde{\mathcal{O}}(\sqrt{n})$-approximations of APSP require time $\widetilde{\Omega}(\sqrt{n})$, which obviously also holds for the CONGEST+NCC model variant considered in this chapter. Furthermore, the lower bound even holds in trees. For the LOCAL+NCC model, the APSP lower bound has been shown to be tight by Schneider and Kuhn [KS20], who gave an $\widetilde{\mathcal{O}}(\sqrt{n})$-time algorithm.

The Diameter Problem, which only requires the nodes to learn the largest distance in G, can be solved more efficiently (see [CLP20] for a comprehensive overview). As Schneider and Kuhn have shown [KS20], there exist graphs in which computing a $(2-\varepsilon)$-approximation of the diameter takes time $\widetilde{\Omega}(n^{1/3})$. Recently, Censor-Hillel et al. [CLP20] came close to this lower bound by presenting a 2-approximation with runtime $\widetilde{\mathcal{O}}(n^{1/3})$. As we demonstrate in this chapter, the lower bound can be avoided by considering sparse graphs.

Censor-Hillel et al. [CLP20] also improve upon various upper bounds for SSSP and related problems. The best currently known (diameter-independent) algorithm for exact SSSP takes time $\widetilde{\mathcal{O}}(n^{1/3})$ [CLP20]. However, for small shortest-path diameter SPD, the $\widetilde{\mathcal{O}}(\sqrt{\mathsf{SPD}})$-time algorithm we present in Chapter 6 is more efficient. Although there does not exist a formal lower bound for SSSP, we believe that a polynomial runtime is required for general graphs, even more so if the local communication is restricted.

A problem closely related to SSSP is the computation of short routing paths between any given nodes. The problem has, for example, been studied in mobile ad hoc networks [Jun+18], in which constant-competitive routing paths can be computed in $\mathcal{O}(\log^2 n)$ rounds [CKS20]. The authors consider a hybrid network model similar to [Aug+20b], where nodes can communicate using either their Wi-Fi interface (similar to the local edges) or the cellular infrastructure (similar to global edges).

Our work also relates to research concerned with *overlay construction* (see Chapter 3 for an overview). More specifically, our algorithms can be used to construct well-formed trees *deterministically* on *pseudotrees* in time $\mathcal{O}(\log n)$, which are graphs that contain at most one cycle. We also heavily use techniques used in overlay networks such as the Euler tour technique or pointer jumping as in Chapter 3. Furthermore, we employ the so-called *shortest-path diameter reduction technique* [Nan14a]. More precisely, by adding shortcuts between nodes in the global network, we can bridge large distances quickly throughout our computations.

As pointed in Chapter 4, the *congested clique* model is somewhat related to the NCC model, which we use to model the global network. Among other problems, shortest paths can be computed remarkably fast in the congested clique [DP20; Cen+19b; Cen+19a]. We highlight the *Multi-Source Shortest Path (MSSP)* algorithm of Dory and Parter [DP20], which computes $(1+\varepsilon)$-approximate shortest path distances from $\mathcal{O}(\sqrt{n})$ sources, and their $(2+\varepsilon)$-approximation for APSP. The runtime of their algorithms is polynomial in $\log\log n$, which is *exponentially* faster than any shortest path algorithm in this thesis. For the Diameter Problem, a $(3/2)$-approximation can be computed in time $\mathcal{O}(\log^2 n/\varepsilon)$ [Cen+19a].

To the best of our knowledge, the best algorithms for *exact* shortest paths in the congested clique still have polynomial runtime [Cen+19b; Le 16; CLT20; Cen+19a]. As an example, APSP can be solved exactly in time $\widetilde{\mathcal{O}}(n^{1/3})$ [Cen+19b], and in time $\mathcal{O}(n^{0.2096})$, if the weights are constant [Le 16]; to the best of our knowledge, there is no algorithm that computes the exact diameter faster than that. For exact SSSP, the best known algorithm has a runtime of $\widetilde{\mathcal{O}}(n^{1/6})$ [Cen+19a].

We remark that most shortest path algorithms for the congested clique rely on efficient matrix multiplications. Our algorithm for sparse graphs also uses matrix multiplication in order to compute APSP between $\mathcal{O}(n^{1/3})$ nodes in the network in $\mathcal{O}(\log^2 n)$ rounds; however, the communication restrictions of the NCC only allow us to multiply matrices with $\mathcal{O}(n^{2/3})$ entries in polylogarithmic time. Unsurprisingly, the much more efficient matrix multiplication techniques for the congested clique are not directly applicable in our model.

By using a simulation framework, we can apply some of the algorithms for PRAMs to our model instead of using native distributed solutions. A formal description of such a simulation is given in the full version of the paper this chapter is based on [FHS20]. For example, we are able to use the algorithms of [DPZ91] to solve SSSP and diameter in trees in time $\mathcal{O}(\log n)$, w.h.p. Furthermore, we can compute the distance between any pair s and t in *outerplanar graphs* in time $\mathcal{O}(\log^3 n)$, w.h.p. by simulating a CREW PRAM. A graph is outerplanar if it admits a planar embedding in which every node lies on the outer boundary, which particularly includes the cactus graphs studied in this chapter. For planar graphs, the distance between s and t can be computed in time $\mathcal{O}(\log^3 n(1+M(q))/n)$, w.h.p., where the nodes know a set of q faces of a planar embedding that covers all vertices, and $M(q)$ is the number of processors required to multiply two $q \times q$ matrices in $\mathcal{O}(\log q)$ time in the CREW PRAM. We remark that all results obtained by simulating PRAMs using our simulation framework are randomized.

For graphs with polylogarithmic arboricity, a $(1+\varepsilon)$-approximation of SSSP can be computed in polylogarithmic time using [Li20] and the PRAM simulation framework (with huge polylogarithmic terms). For general graphs, the algorithm can be combined with well-known spanner algorithms for the CONGEST model (e.g., [BS07]) to achieve constant approximations for SSSP in time $\widetilde{\mathcal{O}}(n^{\varepsilon})$ time in our hybrid model. This yields an alternative to the SSSP approximation algorithm we present in Chapter 6, which also requires time $\widetilde{\mathcal{O}}(n^{\varepsilon})$ but has much smaller polylogarithmic factors.

Contribution and Outline Table 5.1 contains an overview of our results for the SSSP and Diameter Problem and gives an outline of the chapter. The first part

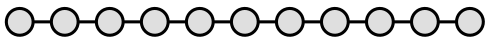
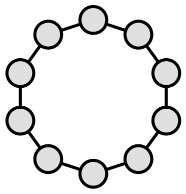
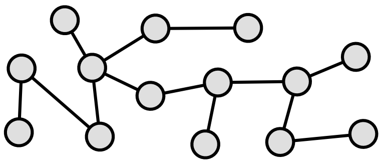
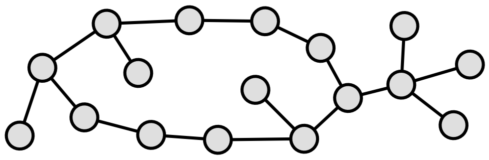
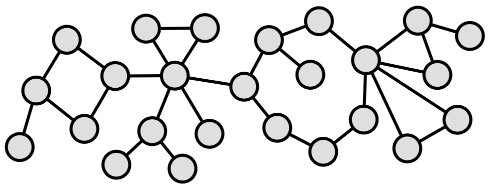
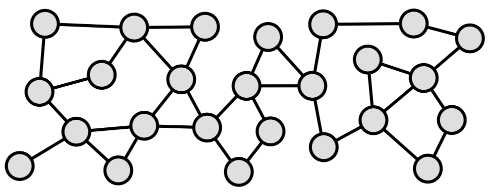
Graph Class	Runtime	Section	Example
Path Graphs (Exact)	$\mathcal{O}(\log n)$	5.1	
Cycle Graphs (Exact)	$\mathcal{O}(\log n)$	5.2	
Trees (Exact)	$\mathcal{O}(\log n)$	5.3	
Pseudotrees (Exact)	$\mathcal{O}(\log n)$	5.4	
Cactus Graphs (Exact)	$\mathcal{O}(\log n)$, w.h.p.	5.5	
Sparse Graphs* (3-approximate)	$\mathcal{O}(\log^2 n)$, w.h.p.	5.6	

Table 5.1.: An overview of the results of this chapter. *Sparse graphs refer to graphs that contain at most $n + \mathcal{O}(n^{1/3})$ edges and have arboricity $\mathcal{O}(\log n)$.

of the chapter revolves around computing SSSP and the diameter on cactus graphs (i.e., connected graphs in which each edge is only contained in at most one cycle). For a comprehensible presentation, we establish the algorithm in several steps. First, we consider the problems in *path graphs* (i.e., connected graphs that contain exactly two nodes with degree 1, and every other node has degree 2), then in *cycle graphs* (i.e., connected graphs in which each node has degree 2), *trees* (i.e., connected graphs that do not contain a cycle), and *pseudotrees* (i.e., connected graphs that contain at most one cycle). For each of these graph classes, we present deterministic algorithms to solve both the SSSP and the Diameter Problem in $\mathcal{O}(\log n)$ rounds, each relying heavily on the results of the previous sections. We then extend our results to *cactus graphs* and present randomized algorithms for SSSP and the diameter with a runtime of $\mathcal{O}(\log n)$, w.h.p. The algorithms rely on a variant of the randomized spanning tree algorithm of Götte et al. [Göt+20].

In Section 5.6, we consider a more general class of sparse graphs, namely graphs with at most $n + \mathcal{O}(n^{1/3})$ edges and arboricity $\mathcal{O}(\log n)$. By using the techniques established in the first part and leveraging the power of the global network to deal with the additional $\mathcal{O}(n^{1/3})$ edges, we obtain algorithms to compute 3-approximations for SSSP and the diameter in time $\mathcal{O}(\log^2 n)$, w.h.p. As a byproduct, we also derive a deterministic $\mathcal{O}(\log^2 n)$-round algorithm for computing a *hierarchical tree decomposition* of the network. The chapter is concluded with an outlook on future work.

We remark that our algorithms do not fully exploit the power of the NCC for the global network. In fact, with little effort all algorithms in this chapter can be adapted to work in the CONGEST+NCC$_0$ model by setting up suitable overlay networks and employing the techniques of Chapter 4. To support this claim, we will occasionally provide some additional details of the necessary adaptions throughout this chapter. Therefore, our algorithms do not only require very little communication, but can also form suitable global network structures as a sideline.

5.1. Path Graphs

To begin with an easy example, we first present a simple algorithm to compute SSSP and the diameter of path graphs. We use the same idea as in Lemma 3.23 of Chapter 3 and perform *pointer jumping* to select a subset of global edges S, which we call *shortcut edges*, that have the following properties: S forms a weighted connected graph with degree $\mathcal{O}(\log n)$ that contains all nodes of V, preserves the distances of the path graph G, i.e., $d_S(u, v) = d_G(u, v)$, and ensures that there is a shortest path between any two nodes in S that contains $\mathcal{O}(\log n)$ hops, i.e., $\mathsf{SPD}(S) = \mathcal{O}(\log n)$. Given such a graph, SSSP can easily be solved by performing a broadcast from s in S for $\mathcal{O}(\log n)$ rounds: In the first round, s sends a message containing $w(e)$ over each edge $e \in S$ incident to s. In every subsequent round, every node $v \in V$ that has already received a message sends a message $k + w(e)$ over each edge $e \in S$ incident to v, where k is the smallest value v has received so far. After $\mathcal{O}(\log n)$ rounds, every node v must have received $d(s, v)$, and cannot have received any smaller value. Further, the diameter of the line can easily be determined by performing SSSP from both of its endpoints u and v, which finally broadcast the diameter $d(u, v)$ to all nodes using the global network.

We construct S using the following simple approach, which in the following we refer to as the *Introduction Algorithm*. S initially contains all edges of E. Additional shortcut edges are established by performing pointer jumping: Every node v first selects one of its at most two neighbors as its *left* neighbor ℓ_1; if it has two neighbors, the other is selected as v's *right* neighbor r_1. Note that the node's notions of left and right do not have to coincide. In the first round of our algorithm, every node v with degree 2 establishes $\{\ell_1, r_1\}$ as a new shortcut edge of weight $w(\{\ell_1, r_1\}) = w(\{\ell_1, v\}) + w(\{v, r_1\})$ by sending the edge to both ℓ_1 and r_1. Whenever at the beginning of some round $i > 1$ a node v with degree 2 receives shortcut edges $\{u, v\}$ and $\{v, w\}$ from ℓ_{i-1} and r_{i-1}, respectively, it sets $\ell_i := u$, $r_i := w$, and establishes $\{\ell_i, r_i\}$ by adding up the weights of the two received edges and informing ℓ_i and r_i. The algorithm terminates after $\lfloor \log(n-1) \rfloor$ rounds. Afterwards, for every simple path in G between u and v with 2^k hops for any $k \leq \lfloor \log(n-1) \rfloor$ we have established a shortcut edge $e \in S$ with $w(e) = d(u, v)$. Therefore, S has the desired properties, and we conclude the following theorem.

Theorem 5.1 (Path Graphs). *SSSP and the diameter can be computed in any path graph in time $\mathcal{O}(\log n)$.*

5.2. Cycle Graphs

In cycle graphs, there are two paths between any two nodes that we need to distinguish. For SSSP, this can easily be achieved by performing the SSSP algorithm for path graphs in both directions along the cycle, and let each node choose the minimum of its two computed distances. Formally, let $v_1, v_2, \ldots, v_n$ denote the n nodes along a *left* traversal of the cycle starting from $s = v_1$ and continuing at s's neighbor of smaller identifier, i.e., $\text{id}(v_2) < \text{id}(v_n)$. For any node u, a shortest path from s to u must follow a left or right traversal along the cycle, i.e., $(v_1, v_2, \ldots, u)$ or $(v_1, v_n, \ldots, u)$ is a shortest path from s to u. Therefore, we can solve SSSP on the cycle by performing the SSSP algorithm for the path graph on $\mathcal{L} := (v_1, v_2, \ldots, v_n)$ and $\mathcal{R} := (v_1, v_n, v_{n-1}, \ldots, v_2)$. Thereby, every node v learns $d_\ell(s, v)$, which is the distance from s to v in $\mathcal{L}$ (i.e., along a left traversal of the cycle), and $d_r(s, v)$, which is their distance in $\mathcal{R}$. It is easy to see that $d(s, v) = \min\{d_\ell(s, v), d_r(s, v)\}$.

Using the above algorithm, s can also easily learn its eccentricity $\text{ecc}(s)$, as well as its *left and right farthest nodes* s_ℓ and s_r. The left farthest node s_ℓ of s is defined as the farthest node v_i along a left traversal of the cycle such that the subpath in $\mathcal{L}$ from $s = v_1$ to v_i is still a shortest path. Formally, $s_\ell = \text{argmax}_{v \in V, d_\ell(s,v) \leq \lfloor W/2 \rfloor} d_\ell(s, v)$, where $W = \sum_{e \in E} w(e)$. The right farthest node s_r is the successor of s_ℓ in $\mathcal{L}$ (or s, if s_ℓ is the last node of $\mathcal{L}$), for which it must hold that $d_r(s, s_r) \leq \lfloor W/2 \rfloor$. We have that $d_\ell(s, s_\ell) = d(s, s_\ell)$, $d_r(s, s_r) = d(s, s_r)$, and $\text{ecc}(s) = \max\{d_\ell(s, s_\ell), d_r(s, s_r)\}$.

To determine the diameter of G, for every node $v \in V$ our goal is to compute $\text{ecc}(v)$; as a byproduct, we will compute v's left and right farthest nodes v_ℓ and v_r. The diameter can then be computed as $\max_{v \in V} \text{ecc}(v)$. A simple way to compute these values is to employ a binary-search style approach from all nodes in parallel, and use the load balancing techniques of Chapter 4 to achieve a runtime of $\mathcal{O}(\log^2 n)$, w.h.p. Coming up with a deterministic $\mathcal{O}(\log n)$ time algorithm, however, is more complicated, and will be the focus of this section.

Our algorithm works as follows. Let s be the node with highest identifier, which is known to all nodes in the NCC, but can also easily be computed by performing

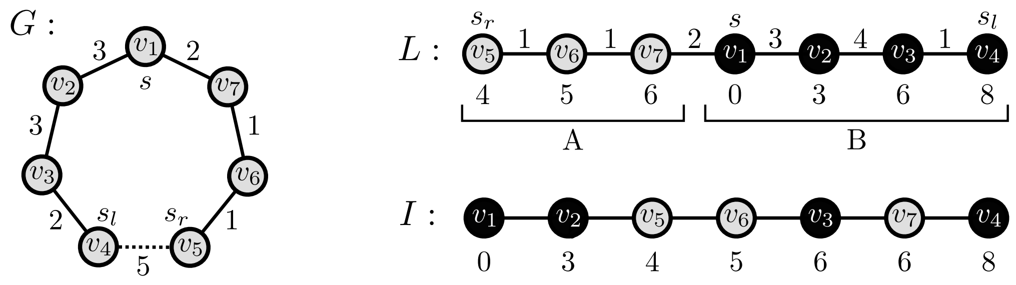


Figure 5.1.: An example of diameter computation in a cycle G. The algorithm begins with $s = v_1$. In L, $s_\ell = v_4$ is the farthest node from s along a left traversal of G, and $s_r = v_5$ is the farthest node along a right traversal. In L, each node is annotated with its budget ϕ. In I, the nodes are sorted by their budget, and for each node $v \in A$ the first node $x \in B$ to its left is its left farthest node v_ℓ. For example, for v_6, $v_\ell = v_2$.

pointer jumping. First, we perform the SSSP algorithm as described above from s in $\mathcal{L}$ and $\mathcal{R}$, whereby s learns its left and right farthest nodes s_ℓ and s_r. Let L be graph that results from removing the edge $\{s_\ell, s_r\}$ from G (see Figure 5.1). Let $A \subseteq V$ be the set of nodes between s_r and s (excluding s), and $B \subseteq V$ be the set of nodes between s and s_ℓ (including s).

In its first execution, our algorithm ensures that each node $v \in A$ learns its left farthest node v_ℓ; a second execution will then handle all other nodes. Note that v_ℓ for all $v \in A$ must be a node of B, since otherwise the path from v to v_ℓ along $\mathcal{L}$ is longer than $\lfloor W/2 \rfloor$, in which case it cannot be a shortest path anymore.

We assign each node v a budget $\phi(v)$, which is $\lfloor W/2 \rfloor - d_r(s, v) \geq 0$, if $v \in A$, and $d_\ell(s, v)$, if $v \in B$. Roughly speaking, the budget of a node $v \in A$ determines how far you can move from v beyond s along a left traversal of G until reaching v's left farthest node v_ℓ. Then, we sort the nodes of L by their budget. Note that since we consider positive edge weights, no two nodes of A and no two nodes of B can have the same budget, but there may be nodes $u \in A$, $v \in B$ with $\phi(u) = \phi(v)$. In this case, we break ties by assuming that $\phi(u) > \phi(v)$. More specifically, the outcome is a sorted list $I = (s = v_{i_1}, v_{i_2}, \ldots, v_{i_n})$ with first node s that contains all nodes of A (and B) in the same order they appear in L, respectively. Such a list can be constructed in time $\mathcal{O}(\log n)$, e.g., by using Aspnes and Wu's algorithm [AW07]. We remark that the algorithm of Aspnes and Wu is actually randomized. However, since we can easily arrange the nodes as a binary tree, we can replace the randomized pairing procedure by a deterministic strategy and still achieve a runtime of $\mathcal{O}(\log n)$; for more details, we refer the reader to [AW07].

Let $v = v_{i_k} \in A$, and let $x = v_{i_j} \in B$ be the node with maximum index j in I such that $j < k$ (i.e., the last node of B in I that is still before v). Since the nodes in I are sorted by their potential, among all nodes of B, x maximizes $\phi(x)$ such that $\phi(x) \leq \phi(v)$. By definition of $\phi(x)$ and $\phi(v)$, this implies that

$$j = \max\{j \in \{1, \ldots, n\} \mid v_{i_j} \in B, j < k \text{ and } d_r(s, v_{i_k}) + d_\ell(s, v_{i_j}) \leq \lfloor W/2 \rfloor\}.$$

Lemma 5.2. *We have that $x = v_\ell$.*

Proof. By the definition of our algorithm, $x \in B$ is the farthest node from $v \in A$ along a left traversal of the cycle such that $d_r(s, v) + d_\ell(s, x) \leq \lfloor W/2 \rfloor$. Note that $d_r(s, v) + d_\ell(s, x) = d_\ell(v, x)$, since s lies between v and x in L. Therefore, x is also farthest from v along a left traversal such that $d_\ell(v, x) \leq \lfloor W/2 \rfloor$, which is the definition of v_ℓ. □

Node v can easily learn x, $d_\ell(s, x)$, and the neighbors of x in the cycle (to infer v_r) by performing the Introduction Algorithm on each connected segment of nodes of A in I. To let all remaining nodes learn their farthest nodes, we restart the algorithm at node s_ℓ as the new source node s. Note that s_ℓ has to perform the algorithm in the same "left direction" as before, which we can easily ensure. Since $d_\ell(s, s_\ell) = d_r(s_\ell, s) \leq \lfloor W/2 \rfloor$, all nodes between s and s_ℓ in $\mathcal{L}$ (except s_ℓ itself), which previously were in set B, will be in set A and learn their farthest nodes. Furthermore, s_ℓ clearly learns its left and right farthest nodes at the beginning of the second execution of our algorithm. We conclude the following theorem.

Theorem 5.3 (Cycle Graphs)**.** *SSSP and the diameter can be computed in any cycle graph G in time $\mathcal{O}(\log n)$.*

5.3. Trees

We now show how the algorithms of the previous sections can be extended to compute SSSP and the diameter on trees. Similar to Lemma 3.23 of Chapter 3, we use the Euler tour technique to transform the graph into a path graph L of *virtual nodes* that corresponds to a depth-first traversal of G (see, e.g., [Gmy+17a]). Every node of G simulates one virtual node for each time it is visited in that traversal, and two virtual nodes are neighbors in L if they correspond to subsequent visitations. To solve SSSP, we assign weights to the edges from which the initial distances in G can be inferred, and then solve SSSP in L instead. Finally, we compute the diameter of G by performing the SSSP algorithm twice, which concludes this section.

However, since a node can be visited up to $\Theta(n)$ times in the traversal, it may not be able to simulate all of its virtual nodes in L. Therefore, we first need to reassign the virtual nodes to the node's neighbors such that every node only has to simulate at most 6 virtual nodes using a Nash-Williams forest decomposition. More precisely, we compute an orientation of the edges in which each node has outdegree at most 3, and reassign nodes according to this orientation (in the remainder of this chapter, we refer to this approach as the *redistribution framework*).

Construction and Simulation of L As in Lemma 3.23, we denote the neighbors of a node $v \in V$ by ascending identifier as $v(0), \dots, v(\deg(v){-}1)$. Consider the depth-first traversal in G that starts and ends at s, and which, whenever it reaches v from some neighbor $v(i)$, continues at v's neighbor $v((i{+}1) \bmod \deg(v))$. L is the directed path graph of virtual nodes that corresponds to this traversal (see Figure 5.2a and 5.2b). The path graph contains a virtual node for each time a node is visited, and a directed edge from each virtual node to its successor in the traversal; however, we

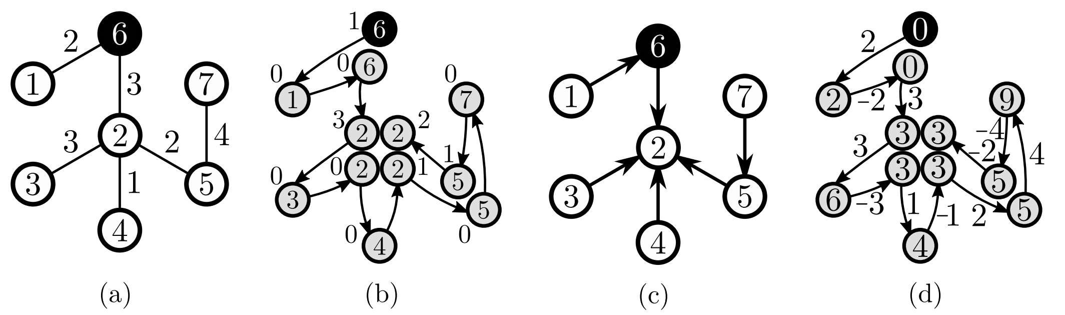


Figure 5.2.: (a) A tree with source node s (black). Each node is labeled with its identifier and each edge is labeled with its weight. (b) The resulting path graph L of virtual nodes. In addition to the identifier of its original node x, each virtual node x_i is annotated with its index i (small numbers at the side). (c) A possible orientation with outdegree 3. According to our redistribution rule, all virtual nodes of the central node 2, for example, would be assigned to its neighbors. (d) The edges are assigned weights, and each virtual node is labeled with its distance to s_L (black node).

leave out the last edge ending at s to break the cycle. More specifically, every node v simulates the nodes $v_0, \dots, v_{\deg(v)-1}$, where v_i corresponds to the traversal visiting v from $v(i)$. The first node of L is $s_L := s_{\deg(s)-1}$, and its last node is the node v_i such that $v = s(\deg(s)-1)$, and $v((i+1) \bmod \deg(v)) = s$. For every node v_i in L (except the last node of L), there is an edge $(v_i, u_j) \in L$ such that $u = v((i+1) \bmod \deg(v))$ and $v = u(j)$. To accordingly introduce each virtual node to its predecessor in L, every node v sends $\mathrm{id}(v_i)$ to $v(i)$ for all $i \in [\deg(v)]$, where $\mathrm{id}(v_i) := \mathrm{id}(v) \circ i$ is the *virtual identifiers* of v_i as in the proof of Lemma 3.23.

It remains to show how the virtual nodes can be redistributed such that each node only has to simulate at most 6 virtual nodes. To do so, we first compute a 3-orientation of G, i.e., an assignment of directions to its edges such that every node has outdegree at most 3; the orientation will help us to assign at most 6 virtual nodes to each node. Since the arboricity of G is 1, we can use [BE10, Theorem 3.5] to compute a partition $V_1, \dots, V_k$ of the nodes such that every node $u \in V_i$ has at most 3 edges leading to nodes in $\bigcup_{j \geq i} V_j$ (see Section 2.2). We obtain our desired orientation by directing each edge $\{u, v\} \in E$, $u \in V_i$, $v \in V_j$, from u to v if $i < j$, or $i = j$ and $\mathrm{id}(u) < \mathrm{id}(v)$ (see Figure 5.2c for an example).

Now consider some node $v \in V$ and a virtual node v_i at v, and let $u := v(i)$. If $\{v, u\}$ is directed from v to u, then v_i is assigned to v, and, as before, v takes care of simulating v_i. Otherwise, v_i gets assigned to u instead, and v sends the identifier of v_i to u. Afterwards, u needs to inform the node w that is responsible for simulating the predecessor of v_i in L that the location of v_i has changed. Depending on the orientation, w is either u itself, a neighbor of u, or a neighbor of one of u's neighbors. In either case, w can be informed about the new location of v_i within at most 2 rounds using the local network. Since in the orientation each node v has at most 3 outgoing edges, for each of which it keeps one virtual node and is assigned one additional virtual node from a neighbor, v has to simulate at most 6 virtual nodes.

We remark that it is possible to combine this technique with the merging algorithm of Lemma 3.23 to establish low-diameter overlays on trees of arbitrary degree without knowing a designated source node beforehand. For instance, this allows us to efficiently compute aggregates of values stored at each tree's nodes, as stated in the following lemma.

Lemma 5.4 (Aggregates in Trees)**.** *Let H be a forest in which every node $v \in V[H]$ stores some value p_v, and let f be a distributive aggregate function. Every node $v \in V$ can learn $f(\{p_u \mid u \in C_v\})$, where C_v is the tree of H that contains v, in time $\mathcal{O}(\log n)$.*

Proof. We let each node $v \in V[H]$ simulate $\deg(v)$ many virtual nodes that correspond to a cyclic depth-first traversal of the component of v as described in Lemma 3.23, and rearrange the nodes using the Nash-Williams forest decomposition technique as described above. Since each node has to simulate at most 6 nodes, the algorithm of Lemma 3.23 can be simulated. As pointed out in Section 3.5, the algorithm can be used to obtain a well-formed tree within each component, in which we can obviously compute aggregates in time $\mathcal{O}(\log n)$. □

Assigning Weights To assign appropriate weights to the edges of L from which we can infer the node's distances in G, we first have to transform G into a rooted tree. To do so, we simply perform SSSP from s_L (the first node in L) in the (unweighted) version of L. Thereby, every virtual node x learns its *traversal distance*, i.e., how many steps the depth-first traversal takes until it reaches x. Further, every node v can easily compute which of its virtual nodes v_i is visited first by taking the minimum traversal distance of its virtual nodes. Since the virtual nodes of v are dispersed among v neighbors in G, the minimum value can be obtained using the local network. Let v_i be the virtual node of v that has smallest traversal distance, and let u_j be the predecessor of v_i in L. It is easy to see that u is the parent of v in the rooted tree, which implies the following lemma.

Lemma 5.5 (Rooting a Tree)**.** *Any tree G can be rooted in $\mathcal{O}(\log n)$ time.*

For each virtual node v_j of v (except the first node s_L), to the edge $(u_i, v_j) \in L$, v assigns the weight

$$w(u_i, v_j) = \begin{cases} w(\{u, v\}) & \text{if } u \text{ is } v\text{'s parent} \\ -w(\{u, v\}) & \text{if } v \text{ is } u\text{'s parent.} \end{cases}$$

An example can be found in Figure 5.2d. Note that if v_j is assigned to a neighbor of v, v needs to informs that neighbor about the weight of the respective edge.

To solve SSSP in G, we simply compute SSSP in (the undirected version of) L using Theorem 5.1. As we prove in the following theorem, the distance of each virtual node v_i of each node v will be $d(s, v)$.

Theorem 5.6 (Tree SSSP)**.** *SSSP can be computed in any tree in time $\mathcal{O}(\log n)$.*

Proof. Let $v \in V$, and let $d_L(s_L, v_i)$ denote the distance from s_L to a virtual node v_i at v in the (weighted) graph L. We show that $d_L(s_L, v_i) = d(s, v)$.

Consider the path P from s to v in G. The depth-first traversal from s to v traverses every edge of P from parent to child, i.e., for every edge in P there is a directed edge with the same weight between s and v_i in L. However, at some of the nodes of P (including s and v) the traversal may take *detours* into other subtrees before traversing the next edge of P. As every edge of L that corresponds to an edge in the subtree is visited, and the weights of all those edges sum up to 0, the distance from s to v_i equals the sum of all edges in P, which is $d(s, v)$. $\square$

Similar techniques lead to the following lemma, which we will use in later sections.

Lemma 5.7 (Computing Subtree Aggregates)**.** *Let H be a forest and assume that each node $v \in V[H]$ stores some value p_v. The goal of each node v is to compute the value $sum_v(u) := \sum_{w \in C_u} p_w$ for each of its neighbors $u \in N_H(u)$, where C_u is the connected component of the subgraph H' of H induced by $V[H] \setminus \{v\}$ that contains u. The problem can be solved in time $\mathcal{O}(\log n)$.*

Proof. Consider a connected component C of H. Since C is a tree, we can determine the node $s \in V[H]$ that has highest identifier among all nodes in $V[C]$ using Lemma 5.4. We construct L exactly as described in the algorithm for computing SSSP with source s on trees, but choose the weights of the edges differently. More precisely, to every edge (u_i, v_j) of L we assign the weight $w(\{u_i, v_j\}) := p_u$, if v is u's parent, and 0, otherwise. Further, we assign a value $\hat{d}(s_L) := p_s$ to s_L (the first node of L). With these values as edge weights, we perform the SSSP algorithm on L from s_L, whereby every virtual node v_i learns the value $\hat{d}(v_i) := \hat{d}(s_L) + d_L(s_L, v_i)$. The sum of all values $M := \sum_{v \in V[H]} p_v$ can be computed and broadcast to every node of C in time $\mathcal{O}(\log n)$ using Lemma 5.4.

The problem can now be solved as follows. Consider some node v, let u be a neighbor of v, and let i be the value such that $u = v(i)$ (recall that $v(i)$ is the neighbor of v that has the i-th highest identifier, $0 \leq i \leq \deg(v) - 1$). If u is the parent of v in the tree rooted at s, then $\sum_{w \in C_u} p_w = M - (\hat{d}(v_{i-1 \bmod \deg v}) - \hat{d}(v_i))$. That is, the result is the sum of all values minus the values of all nodes in the subtree of v (excluding v). If otherwise u is a child of v in the tree rooted at s (unless $v = s$ and $i = \deg(s) - 1$, which is a special case since $s_{deg(s)-1}$ has no incoming edge), then $\sum_{w \in C_u} p_w = \hat{d}(v_i) - \hat{d}(v_{(i-1) \bmod \deg v})$. Finally, if $v = s$ and $i = \deg(s) - 1$, we have $\sum_{w \in C_u} p_w = M - \hat{d}(v_{(i-1) \bmod \deg v})$. $\square$

We remark that using more sophisticated techniques, the previous lemma can be generalized to support the computation of *any* distributive aggregate function. For instance, a randomized solution can be obtained by assigning each shortcut created during pointer jumping the aggregate value of all nodes bridged by the shortcut. The aggregate of each subtree can then be composed from the values of only $\mathcal{O}(\log n)$ shortcuts, which can be retrieved by the root of each subtree using the techniques of Chapter 4 (see [Göt+20] for a detailed description). In fact, using a more complicated distributed load balancing scheme, this approach can even be made deterministic.

Another side result that will be helpful later is given in the following lemma.

Lemma 5.8 (Computing Heights)**.** *Let G be a tree rooted at s. Every node $v \in V$ can compute its height $h(v)$ in G in time $\mathcal{O}(\log n)$.*

Proof. By Theorem 5.6, each leaf node v can learn its distance $d(s,v)$ to s (i.e., its depth in G) in time $\mathcal{O}(\log n)$. For $v = s$, the height is the maximum depth of any node, which can be computed by performing one aggregation using Lemma 5.4. For any other node $v \neq s$, the height $h(v)$ is the maximum depth of any leaf in its subtree minus the depth $d(s,v)$ of v.

Instead of proving the above-mentioned generalization of Lemma 5.4 to obtain the maximum depth of all of v's descendants, we use the following, much simpler, approach. First, every leaf node u assigns its virtual node in L the value $d(s,u)$. We then establish shortcuts on L using the Introduction Algorithm; however, we begin with each edge having the maximum value assigned to any of its endpoints. Whenever a shortcut results from two smaller shortcuts being merged, its weight becomes the maximum weight of the two smaller shortcuts. Thereby, the weight of each shortcut $\{u_i, v_j\}$ (where u_i is the endpoint that is closer to s_L in L) corresponds to the maximum value of any node in G visited by the traversal from u_i to v_j.

Slightly changing notation, let $s_L = x_0, x_1, \ldots, x_{2(n-1)}$ denote all virtual nodes in the order they appear in L (note that the index of each virtual node is its traversal distance, which can easily be computed). Now let $v \neq s$, and x_i be the virtual node of v with smallest, and x_j be its virtual node with highest traversal distance. Let $k = 2^{\lfloor \log(j-i) \rfloor}$. Note that the shortcuts $\{x_i, x_{i+k}\}$ and $\{x_{j-k}, x_j\}$ exist and overlap, and therefore span all virtual nodes of the nodes in the subtree of v in G. Therefore, the value $\max\{w(\{x_i, x_{i+k}\}), w(\{x_{j-k}, x_j\})\}$ gives the maximum depth of any leaf node in v's subtree. Together with the knowledge of $d(s,v)$, v can compute $h(v)$. $\square$

For the diameter, we use the following well-known lemma, which we prove for completeness.

Lemma 5.9. *Let G be a tree, $s \in V$ be an arbitrary node, and let $v \in V$ such that $d(s,v)$ is maximal. Then $ecc(v) = D$.*

Proof. Assume to the contrary that there is a node $u \in V$ such that $\text{ecc}(u) > \text{ecc}(v)$. Then there must be a node $w \in V$ such that $d(u,w) = \text{ecc}(u) > \text{ecc}(v) \geq d(v,w)$. Note that $d(u,w) > d(u,v)$ and $d(u,w) > d(v,w)$, as otherwise $\text{ecc}(u) \leq \text{ecc}(v)$, which would contradict our assumption. Let P_1 be the path from s to v, P_2 be the path from u to w, and let $t = \text{argmin}_{x \in P_2} d(s,x)$ be the node in P_2 that is closest to s, i.e., t is the *lowest common ancestor* of u and w. We distinguish between the cases that $t \notin P_1$ (Case 1) and that $t \in P_1$ (Case 2). For an illustration of these cases, see Figure 5.3.

For Case 1, let x be the node farthest from s that lies on P_1, and also on the path from s to t (x might be s). Then

$$d(u,w) \leq d(u,x) + d(x,w) \leq d(v,x) + d(x,w) = d(v,w),$$

where $d(u,x) \leq d(v,x)$ because v is farthest to s, which contradicts $d(u,w) > d(v,w)$.

In Case 2, i.e., $t \in P_1$, t must lie on the path from v to u or on a path from from v to w. In the first case, $d(u,w) = d(u,t) + d(t,w) \leq d(u,t) + d(t,v) = d(u,v)$, where

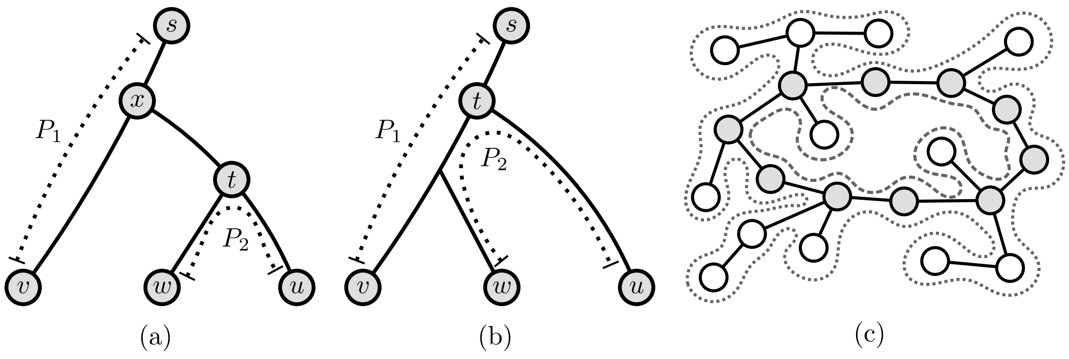


Figure 5.3.: (a) Case 1 of the proof of Lemma 5.9. (b) Case 2 of the proof. This image depicts the case that t lies on the path from v to u; for the case that it lies on a path from v to w, exchange w and u in the image. (c) Example of a pseudotree, in which the Euler tour technique forms two rings. The cycle nodes are gray, and tree nodes are white.

$d(t,w) \leq d(t,v)$ because v is farthest from s, which is a contradiction with $d(u,w) > d(u,v)$. In the second case, we analogously have $d(u,w) = d(u,t) + d(t,w) \leq d(v,t) + d(t,w) = d(v,w) \leq d(v,w)$, which contradicts $d(u,w) > d(v,w)$. $\square$

Therefore, for the diameter it suffices to perform SSSP once from the node s with highest identifier, then choose a node v with maximum distance to s, and perform SSSP from v. Since $\mathrm{ecc}(v) = D$, the node with maximum distance to v yields the diameter. Together with Lemma 5.4, we conclude the following theorem.

Theorem 5.10 (Tree Diameter). *The diameter can be computed in any tree in time $\mathcal{O}(\log n)$.*

5.4. Pseudotrees

Recall that a pseudotree is a graph that contains at most one cycle. We define a *cycle node* to be a node that is part of a cycle, and all other nodes as *tree nodes*. For each cycle node v, we define v's tree T_v as the connected component that contains v in the graph in which v's two adjacent cycle nodes are removed, and denote $h(v)$ as the height of v in T_v. Before we show how SSSP and the diameter can be computed in a pseudotree, we describe how the cycle can be identified, if it exists.

For this, we use the same approach as for the construction of the path L in the tree. We let each node v simulate $\deg(v)$ virtual nodes $v_0, \ldots, v_{\deg(v)-1}$ and connect the virtual nodes according to the same rules as described in Section 5.3, with the exception that we do not leave out the last edge ending at s. If there is no cycle, then this yields a single ring of virtual nodes, in which case we can use our previous algorithms. Otherwise, this will create two rings of virtual nodes with the property that every cycle node must have at least one of its virtual nodes in each virtual ring (see Figure 5.3c for an example). Note that since nodes may have a high degree, we also need to redistribute the virtual nodes using the redistribution framework described in Section 5.3. Since the arboricity of a pseudotree is at most 2, we can compute an orientation with outdegree 6 [BE10, Theorem 3.5], and thus after redistributing the virtual nodes every node simulates at most 12 virtual nodes.

To differentiate the at most two rings of virtual nodes from each other, we first establish shortcuts by using the pointer jumping approach of the Introduction Algorithm. Afterwards, every virtual node broadcasts its virtual identifier along all of its shortcuts; by repeatedly letting each node broadcast the highest identifier received so far for $\mathcal{O}(\log n)$ rounds, each virtual node learns the maximum of all identifiers in its ring. Any node whose virtual nodes learned different maxima must be a cycle node; if there exists no such node, there is no cycle in G. In the NCC, the existence of a cycle node can can easily be determined using the Aggregate-and-Broadcast Algorithm; for the NCC_0, we can perform a variant of Lemma 5.4. We conclude the lemma below.

Lemma 5.11 (Determine the Cycle)**.** *After $\mathcal{O}(\log n)$ rounds every node $v \in V$ knows whether there is a cycle, and, if so, whether it is a cycle node.*

Proof. We argue the correctness of our construction by showing that if G contains one cycle, then (1) the virtual nodes of each tree node are contained in the same virtual ring, (2) each cycle node has two virtual nodes contained in different virtual rings. For (1), let v be a cycle node and $\{v, w\}$ be an edge to some tree node w. By our construction, there is exactly one virtual node v_i of v whose successor is a virtual node of w and there is exactly one virtual node w_i of w whose successor is a virtual node v_j of v. As presented in Section 5.3, this yields a path of virtual nodes starting at v_i that traverses the subtree with root w in a depth-first-search manner and ends at v_j. Therefore, Claim (1) holds for every node in the component of w that results from removing the edge $\{v, w\}$ from G, and, consequently, for all other tree nodes.

Specifically, this shows that the tree nodes do not introduce additional rings to our construction; therefore, we can disregard them and assume that G forms a single cycle that does not contain any tree nodes. For this cycle it has to hold by our construction that every cycle node v has exactly two virtual nodes v_0 and v_1 that are not directly connected to each other. More specifically, the successor of v_0 is a virtual node of one of the two neighbors u of v, and the successor of v_1 is the virtual node of the *other* neighbor w of v. If we follow the directed ring, say, from v_0, then we must visit every node of G in the direction of u before returning to v. However, we cannot reach v via v_1, as the only virtual node of w connected to v_1 is the *successor* of v_1, and cannot be its predecessor, which implies Claim (2). $\square$

Since we already know how to compute SSSP and the diameter on trees, for the remainder of this section we assume that G contains a cycle. In order to solve SSSP, we first perform our SSSP algorithm for tree graphs from source s in the tree T_v in which s lies (note that s may be v itself). Thereby, every node in T_v learns its distance to s. Specifically, the cycle node v learns $d(s, v)$. After performing SSSP with source v on the cycle nodes only, every cycle node $u \neq v$ knows $d(s, v)+d(v, u) = d(s, u)$ and can inform all nodes in its tree T_u about $d(s, u)$ using Lemma 5.4. Finally, u performs SSSP in T_u with source u, whereby each node $w \in T_u$ learns $d(s, u) + d(u, w) = d(s, w)$. Together with Theorems 5.6, we obtain the following theorem.

Theorem 5.12 (Pseudotree SSSP)**.** *SSSP can be computed in any pseudotree in time $\mathcal{O}(\log n)$.*

We now describe how to compute the diameter in a pseudotree. In our algorithm, every cycle node v contributes up to two *candidates* for the diameter. The first candidate for a node v is the diameter of its tree $D(T_v)$. If $\mathrm{ecc}(v) > h(v)$, then v also contributes the value $\mathrm{ecc}(v)+h(v)$ as a candidate. We first show how the values can be computed, and then prove that the maximum of all candidates, which can easily be determined using a variant of Lemma 5.4, is the diameter of G.

After v has identified itself as a cycle node, and knows which of its neighbors are cycle nodes, it can compute its height $h(v)$ in time $\mathcal{O}(\log n)$ using Lemma 5.5 and 5.8 in T_v. Furthermore, $D(T_v)$ can be computed in time $\mathcal{O}(\log n)$ via an application of Theorem 5.10.

It remains to show how v can learn $\mathrm{ecc}(v)$. We define $m_\ell(v) := \max_{u\in V} h(u) - d_\ell(v,u)$, and $m_r(v) := \max_{u\in V} h(u) - d_r(v,u)$. Recall that $d_\ell(v,u)$ and $d_r(v,u)$ denote the distances from v to u along a left or right traversal of the cycle, respectively. Here, we require that the nodes agree on a notion of left and right, which we achieve as a byproduct of the algorithm of Theorem 5.10.

Lemma 5.13 (Eccentricity of Cycle Nodes)**.** *Let $v \in V$ be a cycle node and let v_ℓ and v_r be the left and right farthest nodes of v, respectively. We have that*

$$ecc(v) = \max\{d_\ell(v,v_\ell) + m_r(v_\ell),\, d_r(v,v_r) + m_\ell(v_r)\}.$$

Proof. Let $t \in V$ such that $d(v,t) = \mathrm{ecc}(v)$, and let u be a cycle node such that t is a node of T_u. W.l.o.g., assume that u lies on the *right* side of v, i.e., $d_r(v,u) \le d_\ell(v,u)$. We define d_ℓ and d_r to be $d_\ell(v,v_\ell)$ and $d_r(v,v_r)$, respectively. We show that (1) $d_r + m_\ell(v_r) \ge \mathrm{ecc}(v)$, and that (2) $d_\ell + m_r(v_\ell) \le \mathrm{ecc}(v)$ and $d_r + m_\ell(v_r) \le \mathrm{ecc}(v)$. Both statements together immediately imply the claim.

For (1), note that v_r will consider u as a cycle node for the computation of $m_\ell(v_r)$, and thus $m_\ell(v_r) \ge h(u) - d_\ell(v_r,u)$. Therefore, we have that

$$d_r + m_\ell(v_r) \ge d_r - d_\ell(v_r,u) + h(u) = d_r(v,u) + h(u) = d(v,t).$$

For (2), we only show that $d_\ell + m_r(v_\ell) \le \mathrm{ecc}(v)$; the other side is analogous. Let w be the node such that $m_r(v_\ell) = h(w) - d_r(v_\ell,w)$. First, assume that w lies on the left side of v, i.e., $d_\ell(v,w) \le d_r(v,w)$. In this case, we have that $d_r(v_\ell,w) = d_\ell - d_\ell(v,w)$, which implies

$$\begin{aligned} m_r(v_\ell) &= h(w) - d_r(v_\ell,w) \\ &= h(w) + d_\ell(v,w) - d_\ell \\ &\le \mathrm{ecc}(v) - d_\ell. \end{aligned}$$

Now, assume that w lies on the right side of w, in which case $d_r(v_\ell,w) = d_\ell + d_r(v,w)$. Concluding our proof, we have that

$$\begin{aligned} m_r(v_\ell) &= h(w) - d_r(v_\ell,w) \\ &= h(w) - d_r(v,w) - d_\ell \\ &\le h(w) + d_r(v,w) - d_\ell \\ &\le \mathrm{ecc}(v) - d_\ell. \end{aligned}$$ □

Once each cycle node v knows $m_\ell(v)$ and $m_r(v)$, every cycle node u can easily infer its eccentricity by performing the diameter algorithm for the cycle of Theorem 5.3 to learn its farthest nodes. The corresponding m_ℓ and m_r values can be obtained alongside this execution. Therefore, it remains to show how every cycle node v can compute $m_\ell(v)$ and $m_r(v)$.

To do so, the nodes first establish weighted shortcuts along a left and right traversal using the pointer jumping approach of the Introduction Algorithm on the cycle. Having agreed on a common notion of left and right, the node can ensure that their left and right neighbors correspond to one another (i.e., if u is a right neighbor r_i of v, then v is a left neighbor ℓ_i of u). Afterwards, every cycle node v computes $m_\ell(v)$ (and, analogously, $m_r(v)$) in the following way. v maintains a value x_v, which will obtain the value $m_\ell(v)$ after $\mathcal{O}(\log n)$ rounds. Initially, $x_v := h(v)$. In the first round, every cycle node v sends $x_v - w(\{v, r_1\})$ to its right neighbor r_1. When v receives a value x at the beginning of round i, it sets $x_v := \max\{x_v, x\}$ and sends $x_v - w(\{v, r_i\})$ to r_i.

Lemma 5.14. *At the end of round $\lceil \log n \rceil + 1$, $x_v = m_\ell(v)$.*

Proof. We show that at the end of round $i \geq 1$,

$$x_v = \max_{u \in V_\ell(v,i)} (h(u) - d_\ell(v,u)),$$

where $V_\ell(v,i)$ contains node $u \in V$ if the (directed) path from v to u in G_ℓ contains at most $2^{i-1} - 1$ hops. The lemma follows from the fact that $V_\ell(v, \lceil \log n \rceil + 1) = V$.

At the end of round 1, $x_v = h(v)$, which establishes the base case since v is the only node within 0 hops from v. By the induction hypothesis, at the beginning of round $i > 1$ we have that $x_v = \max_{u \in V_\ell(v,i-1)}(h(u) - d_\ell(v,u))$. Furthermore, v receives

$$\begin{aligned} x &= \max_{u \in V_\ell(\ell_{i-1}, i-1)} (h(u) - d_\ell(\ell_{i-1}, u)) - w(\{\ell_{i-1}, v\}) \\ &= \max_{u \in V_\ell(\ell_{i-1}, i-1)} (h(u) - d_\ell(v,u)). \end{aligned}$$

Since $V_\ell(v, i-1) \cup V_\ell(\ell_{i-1}, i-1) = V_\ell(v,i)$, we conclude that

$$\max\{x_v, x\} = \max_{u \in V_\ell(v,i)} (h(u) - d_\ell(v,u)). \qquad \square$$

Using the previous results, the nodes can now compute their candidates and determine the maximum of all candidates. It remains to show the maximum of all candidates actually gives the diameter of G.

Lemma 5.15. *Let C be the set of all candidates.* $\max_{c \in C}\{c\} = D$.

Proof. First, note that since every candidate value corresponds to the length of a shortest path in G, $c \leq D$ for all $c \in C$. Let $s, t \in V$ be two nodes such that $D = d(s,t)$, and let T_v and T_w with cycle nodes v and w be the trees of s and t, respectively. We show that v or w chooses D as one of their candidates. First, note

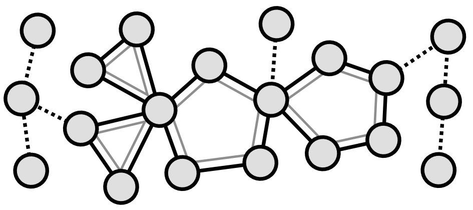


Figure 5.4.: The blocks of a cactus graph. Each block is either a single edge (dotted edges), or a simple cycle (outlined by gray edges).

that if one of the two nodes s and t, say s, is a cycle node, then $D = \text{ecc}(v) = \text{ecc}(v) + h(v)$, and $\text{ecc}(v) > h(v) = 0$; therefore, v chooses D as a candidate.

Therefore, assume that both s and t are tree nodes. If s and t belong to the same tree, i.e., $v = w$, we have that $d(s, t) = D(T_v)$, which is a candidate of v. Otherwise, we only have to show that $\text{ecc}(v) > h(v)$ or $\text{ecc}(w) > h(w)$, since either case implies that v or u contributes $d(s, t)$ as a candidate. Assume to the contrary that $\text{ecc}(v) = h(v)$ and $\text{ecc}(w) = h(w)$ (note that $\text{ecc}(u) \geq h(u)$ for every cycle node u). Since $\text{ecc}(w) \geq h(v)$, we clearly have that $\text{ecc}(v) = h(v) \leq \text{ecc}(w) = h(w)$. However, since v and w are different nodes, we also have that $\text{ecc}(v) \geq d(v, w) + h(w) > h(w)$, which is a contradiction. □

We conclude the final theorem of this section.

Theorem 5.16 (Pseudotree Diameter)**.** *The diameter can be computed in any pseudotree in time $\mathcal{O}(\log n)$.*

5.5. Cactus Graphs

Our algorithms for cactus graphs rely on an algorithm to compute the *biconnected components* (or *blocks*) of G. A block is a maximal biconnected subgraph, where a graph is called *biconnected* if the removal of a single node would not disconnect the graph. Note that for any graph, each edge lies in exactly one block. In case of cactus graphs, each block is either a single edge that is not contained in any cycle (a *bridge*) or a simple cycle (see Figure 5.4). By computing the blocks of G, each node $v \in V$ classifies its incident edges into bridges (if there is no other edge incident to v contained in the same block) and pairs of edges that lie in the same cycle.

Computing the Blocks of G We compute the blocks by using a variant of the algorithm of Götte et al. [Göt+20, Theorem 1.3] for the NCC_0 under the constraint that the input graph (which is not necessarily a cactus graph) has constant degree. We point out how the lemma is helpful for cactus graphs, and then use a simulation of the biconnectivity algorithm of Tarjan and Vishkin [TV85] akin to the approach of Götte et al. [Göt+20, Theorem 1.4] to compute the blocks of G. The description and proofs of the following three lemmas are very technical and mainly describe adaptions of the algorithm of Götte et al.

Lemma 5.17 (Variant of [Göt+20, Theorem 1.3])**.** *Let G be any graph with constant degree. A spanning tree of G can be computed in time $\mathcal{O}(\log n)$, w.h.p., in the NCC_0.*

Proof. To prove the lemma, we need a combination of [Göt+20, Theorem 1.1], which transforms the initial graph into a well-formed tree, w.h.p., and a variant of the spanning tree algorithm given in [Göt+20, Theorem 1.3].

In [Göt+20, Theorem 1.1], a well-formed tree is obtained from an intermediate graph G_L that has degree and diameter $\mathcal{O}(\log n)$. The edges of G_L are created by performing random walks of constant length in a graph G_{L-1}, which again is obtained from random walks in G_{L-2}, where G_0 is the graph G extended by some self-loops and edge copies. More precisely, each edge of G_i results from performing a random walk of constant length in G_{i-1}, and connecting the two endpoints of the random walk.

A spanning tree of G is obtained as follows. Let B be a BFS tree of G_L, which can be computed by a simple breadth-first search since G_L that has degree and diameter $\mathcal{O}(\log n)$. Consider a depth-first traversal of B that is rooted at the node with highest identifier. By using the Euler tour technique as described in Section 5.3, each node can determine the edge of B over which it is reached *first* in the traversal (we can apply the technique since G_L has degree $\mathcal{O}(\log n)$, and can therefore use the redistribution framework using communication in the NCC_0 only). Furthermore, we can enumerate the edges in the order they are visited by the traversal. We then replace each edge of B by the edges of G_{L-1}. Since each edge of B resulted from a random walk of constant length in G_{L-1}, and each node is traversed by at most $\mathcal{O}(\log n)$ random walks in each graph, the nodes of each edge can be informed in constant time. Specifically, each node can easily infer the edge of G_{L-1} over which it is reached first in the depth-first traversal of B if each edge was replaced by edges of G_{L-1} (see also Theorem 3.31). The union of these first edges in G_{L-1} gives a spanning tree of G_{L-1}. By repeating this process, after $\mathcal{O}(\log n)$ steps we have a spanning tree of G_0, which must also be a spanning tree of G. □

Now let G be a cactus graph. To use the previous lemma on cactus graphs, we need to transform G into a constant-degree graph G' using the idea of Section 3.2 in Chapter 3. Then, we compute a spanning tree S' on G' using the previous lemma, and finally infer a spanning tree S of G from S' using the approach of Section 3.5. The details can be found in the proof of the following lemma.

Lemma 5.18 (Cactus Graph Spanning Tree). *A spanning tree of a cactus graph G can be computed in time $\mathcal{O}(\log n)$, w.h.p.*

Proof. We first use [BE10, Theorem 3.5] to compute an orientation of G; since the arboricity of G is 2, each node v will have 6 out-neighbors. By arranging the in-neighbors of each node as a list as described in Section 3.2, Step 2, we obtain a graph G' with degree at most 13 (i.e., each node keeps at most 6 outgoing edges, is assigned at most 6 new siblings, and keeps at most one incoming edge). By using the algorithm of Lemma 5.17, we obtain a spanning tree S' of G'. To infer a spanning tree S of G, we use the algorithm of Theorem 3.31 that first enumerates the nodes of S' along a depth-first traversal, and then selects edges in G according to this enumeration. As described for Theorem 3.31, this yields a spanning tree of G. □

To obtain the blocks of G, we perform a simulation of the algorithm of Tarjan and Vishkin [TV85] in almost the same way as Götte et al. [Göt+20, Theorem 1.4]. The algorithm relies on a spanning tree, which we compute using Lemma 5.18, and constructs a *helper graph* whose connected components yield the blocks of G. The only difference from our application to the simulation described by Götte et al. lies in the fact that we do not rely on the complicated algorithm of [Göt+20, Theorem 1.2] for computing the connected components, but use Lemma 5.17 together with the child-sibling approach of Section 3.2, exploiting the properties of a cactus graph.

Lemma 5.19 (Cactus Graph Blocks)**.** *The blocks of a cactus graph G can be computed in time $\mathcal{O}(\log n)$, w.h.p.*

Proof. Let S be a spanning tree of G rooted at the node with highest identifier s obtained by using Lemmas 5.18 and 5.5, where s is known from the algorithm of Lemma 5.17. In the biconnectivity algorithm of Tarjan and Vishkin [TV85], a helper graph G'' is constructed that contains a node $\{u, v\}$ for every edge $\{u, v\}$ of S, and nodes are connected according to three rules. We use the simulation described in [Göt+20, Section 4.4], where every node $v \neq s$ simulates the edge to its parent u, and the edges of G'' are determined by computing some subtree aggregates using a variant of Lemma 5.7.

The only difference lies in Step 4 of the simulation. Instead of using [Göt+20, Theorem 1.2] to compute the connected components of the helper graph G'', we observe that every node v is only adjacent to at most one node w in G that lies in a different subtree than v in the rooted spanning tree S of G; otherwise, the edge from v to its parent u in S would lie in two different cycles, which is not possible in a cactus graph. Therefore, the edge $\{v, u\}$ in S (which becomes a node in G'') will only create *one* connection to the parent edge $\{w, x\}$ of w according to Rule 1 of Step 3 of [Göt+20, Section 4.4]. Rules 2 then adds an additional single edge to each node of G'', and the connections of Rule 3 can be disregarded completely for our simulation.

Therefore, the arboricity of G'' is constant, and we can again (1) use [BE10, Theorem 3.5] to compute an $\mathcal{O}(1)$-orientation of G'', (2) transform the graph into a constant-degree graph using the child-sibling approach of Section 3.2, and (3) compute a spanning tree on each connected component of the resulting graph using Lemma 5.17. As Götte et al. [Göt+20] point out, the simulation of these algorithms is possible since the local communication required for (1) and (2) can be carried out using a constant number of local communication rounds in G, and for (3) every node $v \in V$ only simulates a single node with constant degree. Finally, using a simulation of Lemma 5.4 on the spanning trees, we can distinguish the connected components from one another, which concludes the lemma. □

SSSP in Cactus Graphs Using the lemma above, every node can determine which of its incident edges lie in the same block in time $\mathcal{O}(\log n)$, w.h.p. Let s be the source for the SSSP Problem. First, we compute the *anchor node a_C* of each cycle C in G, which is the node of the cycle that is closest to s (if s is a cycle node, then the anchor node of that cycle is s itself). To do so, we replace each cycle C in G by a binary tree T_C of height $\mathcal{O}(\log n)$ as described in [Gmy+17a]. More precisely, we first establish shortcut edges using the Introduction Algorithm in each cycle, and then perform a broadcast from the node with highest identifier in C for $\mathcal{O}(\log n)$ rounds. If in some round a node receives the broadcast for the first time from ℓ_i or r_i, it sets that node as its parent in T_C and forwards the broadcast to ℓ_j and r_j, where $j = \min\{i - 1, 0\}$. After $\mathcal{O}(\log n)$ rounds, T_C is a binary tree that contains all nodes of C and that has height $\mathcal{O}(\log n)$. To perform the execution in all cycles in parallel, each node simulates one virtual node for each cycle it lies in and connects the virtual nodes using its knowledge of the blocks of G. To keep the global communication low, we again use the redistribution framework described in Section 5.3, which is possible since the arboricity of G is 2.

Lemma 5.20 (Anchor Nodes)**.** *Let T be the (unweighted) tree that results from taking the union of all trees T_C and all bridges in G. For each cycle C, the node* $\operatorname{argmin}_{v \in C} d_T(s, v)$ *is the anchor node a_C of C.*

The correctness of the lemma above simply follows from the fact that any shortest path from s to any node in C must contain the anchor node of C both in G and in T. Therefore, the anchor node of each cycle can be computed by first performing the SSSP algorithm for trees with source s in T and then conducting a broadcast in each cycle. Now let v be the anchor node of some cycle C in G. By performing the diameter algorithm of Theorem 5.3 in C, v can compute its left and right farthest nodes v_ℓ and v_r in C. Again, to perform all executions in parallel, we use our redistribution framework. Our approach for SSSP is based on the following lemma.

Lemma 5.21 (Shortest-Path Tree)**.** *Let S be the graph that results from removing the edge $\{v_\ell, v_r\}$ from each cycle C with anchor node v. S is a shortest-path tree of G with source s.*

Proof. Since we delete one edge of each cycle, and every edge is only contained in exactly one cycle, S is a tree. Assume to the contrary that S does not contain a shortest path from s to some other node, i.e., there exists a path P from s to some node t in G that is shorter than the (unique) shortest path P_S from s to t in S. Clearly, if P traverses any cycles, then P_S traverses these cycles in the same order, and each cycle is entered and left over the same nodes in P and P_S. However, P may differ from P_S in that it may traverse some cycles over the *other side* of the cycle, traversing the deleted edge. Since $w(P) < w(P_S)$, not *all* subpaths over cycles that traverse the deleted edge of that cycle can have the same length as the subpath that goes along the other side of the respective cycle.

More specifically, P must contain an edge e of a cycle C such that the subpath $P' = (v, \dots, u)$ of P that contains only the nodes of C is strictly shorter than the subpath P'_S of P_S from v to u in S. Note that v must be the anchor node of C, and P' is the (unique) path from v to u in C that contains e, whereas S contains the unique path P'_S from v to u that does *not* contain e (i.e., it goes along the other side of the cycle). However, by definition of e, P'_S must already be a shortest path between v and u in G, which contradicts the assumption that P' is shorter. □

Note that the shortest-path tree of the above lemma can easily be rooted at s using Lemma 5.5. Therefore, we can perform the SSSP algorithm for trees given in Theorem 5.6 on S. We conclude the following theorem.

Theorem 5.22 (Cactus Graph SSSP)**.** *SSSP can be computed in any cactus graph in time $\mathcal{O}(\log n)$, w.h.p.*

Diameter in Cactus Graphs To compute the diameter of G, we first perform the algorithm of Lemma 5.21 with the node that has highest identifier as source s, which yields a (rooted) shortest-path tree S. Note that s is known, e.g., from the execution of the algorithm of Lemma 5.17. Let $Q(v)$ denote the children of v in S. Using Lemma 5.8, each node v can compute its height $h(v)$ in S and can locally determine the value

$$m(v) := \max_{u,w \in Q(v), B(u) \neq B(w)} (h(u) + h(w) + w(v,u) + w(v,w)),$$

where $B(u)$ denotes the block of u. Note that $m(v)$ corresponds to the length of some shortest path of G.

We further define the pseudotree Π_C of each cycle C as the graph that contains all edges of C and, additionally, an edge $\{v, t_v\}$ for each node $v \neq a_C$ of C, where t_v is a node that is simulated by v, and $w(\{v, t_v\}) = \max_{u \in Q(v) \setminus C}(h(u) + w(\{v,u\}))$. Intuitively, each node v of C that is not the anchor node is attached an edge whose weight equals the height of its subtree in S without considering the child of v that also lies in C (if that exists). Then, for each cycle C in parallel, we perform the algorithm of Theorem 5.16 on Π_C to compute its diameter $D(\Pi_C)$ (using the redistribution framework). We obtain the diameter of G as the value

$$\hat{D} := \max\{h(s),\ \max_{v \in V} m(v),\ \max_{\text{cycle } C} D(\Pi_C)\}.$$

By showing that $\hat{D} = D$, we conclude the following theorem.

Theorem 5.23 (Cactus Graph Diameter)**.** *The diameter can be computed in any cactus graph in time $\mathcal{O}(\log n)$, w.h.p.*

Proof. We first show that $\hat{D} \leq D$, and then prove $D \leq \hat{D}$.

First, note that $h(v) \leq D$ for all $v \in V$. Now let $v \in V$ and consider $h(u) + h(w) + w(v,u) + w(v,w)$ for some $u, w \in Q(v)$ such that $B(u) \neq B(w)$. Clearly, $h(u)$ and $h(w)$ correspond to the length of a shortest path since S is a shortest-path tree and u and w are children of v in S. Furthermore, since the edges $\{v,u\}$ and $\{v,w\}$ lie in different blocks, there is no other path between nodes u and w apart from going over v. This implies that $m(v) \leq D$. Finally, the weight of each attached edge $\{v, t_v\}$ of a cycle C corresponds to the length of the path from v to some descendant of v in S, which must be a shortest path. Therefore, for any shortest path in Π_C, a shortest path of the same length must exist in G, which implies that $D(\Pi_C) \leq D$. We conclude that $\hat{D} \leq D$.

To show that $D \leq \hat{D}$, let $P = (v_1, \ldots, v_k)$ be a longest shortest path in G. First, note that each cycle in G is only entered and left at most once by P (if it is left at some node, it may only be entered again at the same node, which is impossible since we have positive edge weights). We first slightly change P to ensure that it does not simultaneously contain the deleted edge and the anchor node of the same cycle. Consider any (maximal) subpath P_C of P that contains edges of a cycle C in G, and assume that P' contains the deleted edge e_C of C (i.e., the edge that is incident to the farthest nodes of C's anchor node a_C in the cycle). If P_C contains both e_C and a_C, then the cycle must be *symmetric*: P_C begins at e_C and ends at a_C

(or vice versa), and the other side of the cycle has the same weight as P_C; any other case would contradict the definition of e_C. Therefore, we can simply replace P_C by the other edges of C and obtain a path of the same weight that does not contain e_C. After replacing each subpath P_C of P for each cycle C, P either (1) only contains edges of S, or (2) contains a deleted edge e_C. In the second case, P cannot contain a_C, and therefore every other cycle contained in P must be entered over its anchor node, which implies that no other deleted edge can be contained in P.

Assume that Case (1) holds. In this case, P can be divided into two paths $P_1 = (v_1, \dots, v_j)$, where $\{v_i, v_{i+1}\}$ is directed from v_i to v_{i+1} for all $1 \leq i < j$, and $P_2 = (v_j, \dots, v_k)$, where $\{v_i, v_{i+1}\}$ is directed from v_{i+1} to v_i for all $j \leq i < k$. If either P_1 or P_2 is empty, then $v_j = s$ and $D = h(s) \leq \hat{D}$. Otherwise, if the edges $\{v_{j-1}, v_j\}$ and $\{v_j, v_{j+1}\}$ lie in different blocks, $D = w(P_1) + w(P_2) \leq m_{v_j} \leq \hat{D}$. If they lie in the same block (more precisely, in the same cycle C), then there exists a corresponding shortest path in Π_C, and $D \leq D(\Pi_C) \leq \hat{D}$.

Finally, assume that Case (2) holds. Let e_C be the single deleted edge of P and P_C be the subpath of P contained in C. By the arguments above, P_C does not contain a_C. Let u and v be the two endpoints of P_C, where u is visited in P before v, and let $P = P_u \circ P_C \circ P_v$, where P_u is the subpath from P's first node to u, and P_v is the subpath from v to the last node to P. Since P_u and P_v do not contain any deleted edge, nor any edge of C, it holds that $w(P_u) \leq w(\{u, t_u\})$ (recall that the weight of u's virtual edge is the height of u in S without the subtree of u's child in C), and $w(P_v) \leq w(\{v, t_v\})$. Let P' be the path in Π_C that starts at $\{u, t_u\}$, then follows P_C, and ends at $\{v, t_v\}$. Since P_C is a shortest path in C, P' is a shortest path in Π_C and $w(P) \leq w(P')$, which implies that $D = w(P) \leq w(P') \leq D(\Pi_C) \leq \hat{D}$. $\square$

5.6. Sparse Graphs

In this final section, we present constant factor approximations for SSSP and the diameter in graphs that contain at most $n + \mathcal{O}(n^{1/3})$ edges and that have arboricity at most $\mathcal{O}(\log n)$. Our algorithm for such graphs relies on an MST $M = (V, E')$ of G, where $E' \subseteq E$. M can be computed deterministically in time $\mathcal{O}(\log^2 n)$ using the algorithm of Gmyr et al. [Gmy+17a, Observation 4] in a modified way, as described in the lemma below. We remark that the original algorithm computes a (not necessarily minimum) spanning tree, which would actually already suffice for the results of this chapter. However, if G contains edges with exceptionally large weights, an MST may yield much better results in practice.

Lemma 5.24 (MST). *There is an algorithm that computes an MST of G deterministically in time $\mathcal{O}(\log^2 n)$.*

Proof. Similar to our algorithm of Chapter 3, the *Overlay Construction Algorithm* presented by Gmyr et al. [Gmy+17a] constructs a well-formed tree in time $\mathcal{O}(\log^2 n)$ by alternatingly grouping and merging supernodes until a single supernode remains. As a byproduct, as remarked in [Gmy+17a, Observation 4], the edges over which merge requests have been sent from one supernode to another form a spanning tree.

To obtain an MST, we change the way a supernode u chooses a neighboring supernode to merge with. More specifically, we use the same approach as for our

MST algorithm for the NCC in Section 4.2 and mimic Borůvka's algorithm. Instead of choosing the adjacent supernode v that has the highest identifier, and sending a merge request if v's identifier is higher than u's identifier, u simply picks its lightest outgoing edge. Unlike the grouping stage described by Gmyr et al., this yields components of supernodes that form pseudotrees with a cycle of length 2 (see, e.g., [JM95]). However, such cycles can easily be resolved locally by the supernodes such that the resulting components form trees.

To perform the merging stage of the algorithm in our model, in which each supernode becomes internally reorganized as a binary tree of depth $\mathcal{O}(\log n)$, we can use a combination of our previous techniques. First, we transform the internal spanning tree of each component, which consists of all previously selected merge edges, into a constant-degree *child-sibling tree*. To do so, we first root the tree using Lemma 5.5, and then let each inner node arrange its children as a list, keeping an edge only to its first child (see also Section 3.2). Using the strategy of Lemma 3.23, we can easily establish a binary tree within each component. □

We call each edge $e \in E \setminus E'$ a *non-tree edge*. Further, we call a node *shortcut node* if it is adjacent to a non-tree edge, and define $\Sigma \subseteq V$ as the set of shortcut nodes. Contrary to the MST algorithm of Section 4.2, after computing M every node $v \in \Sigma$ knows that it is a shortcut node, i.e., if one of its incident edges has not been added to E'. In the remainder of this section, we will compute approximate distances by (1) computing the distance from each node to its closest shortcut node in G, and (2) determining the distance between any two shortcut nodes in G. For any $s, t \in V$, we finally obtain a good approximation for $d(s,t)$ by considering the path in M as well as a path that contains the closest shortcut nodes of s and t.

Our algorithms rely on a *balanced decomposition tree* T_M, which allows us to quickly determine the distance between any two nodes in G, and which is presented in Section 5.6.1. In Section 5.6.2, T_M is extended by a set of edges that allow us to solve (1) by performing a distributed multi-source Bellman-Ford algorithm for $\mathcal{O}(\log n)$ rounds. For (2), in Section 5.6.3 we first compute the distance between any two shortcut nodes in M, and then perform matrix multiplications to obtain the pairwise distances between shortcut nodes in G. By exploiting the fact that $|\Sigma| = \mathcal{O}(n^{1/3})$, and using techniques of Chapter 4, we are able to distribute the $\Theta(n)$ operations of each of the $\mathcal{O}(\log n)$ multiplications efficiently using the global network. In Section 5.6.4, we finally show how the information can be used to compute 3-approximations for SSSP and the diameter.

For simplicity, in the following sections we assume that M has degree 3. Justifying this assumption, we remark that M can easily be transformed into such a tree while preserving the distances in M: First, we root the tree at the node with highest identifier using Lemma 5.5. Then, every node v replaces the edges to its children by a binary tree of virtual nodes, where the leaf nodes are the children of v, the edge from each leaf u to its parent is assigned the weight $w(\{v,u\})$, and all inner edges have weight 0. The virtual nodes are distributed evenly among the children of v such that each child is only tasked with the simulation of at most one virtual node. Clearly, the virtual edges can be established using the local network. Furthermore, since the resulting graph has constant degree, any algorithm in this section that relies on local communication can actually be performed using global communication only. Note that the edge weights of this construction are no longer strictly positive; however, one can easily verify that the algorithms of this section also work with non-negative edge weights.

5.6.1. Hierarchical Tree Decomposition

In this section, we present an algorithm to compute a hierarchical tree decomposition of M, resulting in a *balanced decomposition tree* T_M. T_M will enable us to compute distances between nodes in M in time $\mathcal{O}(\log n)$, despite the fact that the diameter of M may be very high.

Our algorithm constructs T_M as a binary rooted tree $T_M = (V, E_T)$ of height $\mathcal{O}(\log n)$ with root $r \in V$ (which is the node that has highest identifier) by selecting a set of global edges E_T. Each node $v \in V$ knows its parent $p_T(u) \in V$. To each edge $\{u, v\} \in E_T$ we assign a weight $w(\{u, v\})$ that equals the sum of the weights of all edges on the (unique) path from u to v in M. Further, each node $v \in V$ is assigned a distinct label $l(v) \in \{0,1\}^{\mathcal{O}(\log n)}$ such that $l(v)$ is a prefix of $l(u)$ for all children u of v in T_M, and $l(r) = \varepsilon$ (the empty word).

From a high level, the algorithm works as follows. Starting with M, within $\mathcal{O}(\log n)$ iterations M is divided into smaller and smaller components until each component consists of a single node. More specifically, in iteration i, every remaining component A handles one *recursive call* of the algorithm, where each recursive call is performed independently from the recursive calls executed in other components. The goal of A is to select a *split node* x, which becomes a node at depth $i-1$ in T_M, and whose removal from M divides A into components of size at most $|A|/2$. The split node x then recursively calls the algorithm in each resulting component; the split nodes that are selected in each component become children of x in T_M (see the example in Figure 5.5).

When the algorithm is called at some node v, it is associated with a *label* parameter $l \in \{0,1\}^{\mathcal{O}(\log n)}$ and a *parent* parameter $p \in V$. The first recursive call is initiated at node r with parameters $l = \varepsilon$ and $p = \emptyset$. Assume that a recursive call is issued at $v \in V$, let A be the component of M in which v lies, and let A_1, A_2 and A_3 be the at most three components of A that result from removing v. Using Lemma 5.7, every node u in A_1 can easily compute the number of nodes that lie in each of its adjacent subtrees in A_1 (i.e., the size of the resulting components of A_1 after removing u). There must exist a *split node* x_1 in A_1 whose removal divides A_1 into components of size at most $|V[A_1]|/2$ (the existence of such a node is shown in Lemma 6.3 of the following chapter); if there are multiple such nodes, let x_1 be the one that has highest identifier. Correspondingly, there are split nodes x_2 in A_2 and x_3 in A_3. v learns x_1, x_2 and x_3 using Lemma 5.4 and sets these nodes as its children in T_M. By performing the SSSP algorithm of Theorem 5.6 with source v in A_1, x_1 learns $d_M(x_1, v)$, which becomes the weight of the edge $\{v, x_1\}$ (correspondingly, the edges $\{v, x_2\}$ and $\{v, x_3\}$ are established). To continue the recursion in A_1, x calls x_1 with label parameter $l \circ 00$ and parent parameter v. Additionally, x_2 is called with $l \circ 01$, and x_3 with $l \circ 10$.

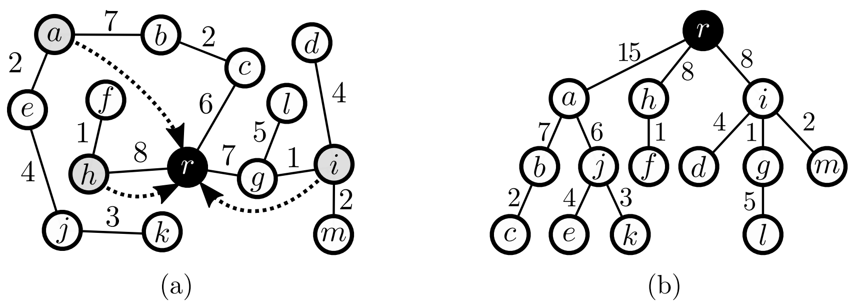


Figure 5.5.: (a) The graph M after the first step of the tree decomposition. The black node is the root r, and the gray nodes are the first split nodes chosen for each of r's subtrees. The algorithm will recursively be called in each connected component of white nodes. (b) A possible resulting balanced decomposition tree T_M.

Theorem 5.25 (Balanced Decomposition Tree)**.** *A balanced decomposition tree T_M for M can be computed in time $\mathcal{O}(\log^2 n)$.*

Proof. It is easy to see that our algorithm constructs a correct balanced decomposition tree. It remains to analyze the runtime of our algorithm. In each recursive call we need $\mathcal{O}(\log n)$ rounds to compute the sizes of all subtrees for any node by Lemma 5.7 and $\mathcal{O}(\log n)$ rounds to find a split node due to Lemma 5.4. Computing the weight of a global edge chosen to be in E_T takes $\mathcal{O}(\log n)$ rounds by Theorem 5.6. Since the component's sizes at least halve in every iteration, the algorithm terminates after $\mathcal{O}(\log n)$ iterations. This proves the theorem. □

Clearly, one can route a message from any node s to any node t in $\mathcal{O}(\log n)$ rounds by following the unique path in the tree from s to t, using the node labels to find the next node on the path. However, the sum of the edge weights along that path may be higher than the actual distance between s and t in M.

5.6.2. Finding Nearest Shortcut Nodes

To efficiently compute the nearest shortcut node for each node $u \in V$, we extend T_M to a *distance graph* $D_T = (V, E_D)$, $E_D \supseteq E_T$, by establishing additional edges between the nodes of T_M. Specifically, unlike T_M, the distance between any two nodes in D_T will be equal to their distance in M, which allows us to employ a distributed Bellman-Ford approach.

We describe the algorithm to construct D_T from the perspective of a fixed node $u \in V$ (for an illustration, see Figure 5.6). For each edge $\{u, v\} \in E_T$ such that $u = p_T(v)$ for which there does *not* exist a local edge $\{u, v\} \in E'$, we know that the edge $\{u, v\}$ "skips" the nodes on the unique path between u and v in M. Consequently, these nodes must lie in a subtree of v in T_M. Therefore, to compute the exact distance from u to a skipped node w, we cannot simply add up the edges in E_T on the path from u to w, as this sum must be larger than the distance $d(u, w)$.

To circumvent this problem, u's goal is to establish additional edges to some of these skipped nodes. Let $x \in V$ be the neighbor of u in M that lies on the

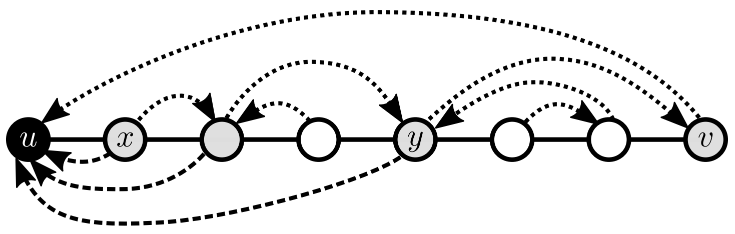


Figure 5.6.: Example for the construction of additional edges (indicated by the dashed lines) going into the node u. Straight black edges are edges of M, and dotted edges are the edges of T_M, directed from child to parent. v is a child of u in T_M, and x is the descendant of v adjacent to u in M. Starting with v's child y in T_M, every descendant of v that goes in the direction of u in M adds an edge to u in D_T (dashed edges below). No white node adds an edge to u, since it lies in the direction opposite to u.

unique path from u to v in M. To initiate the construction of edges in each of its subtrees, u sends a message to each child v of its at most 3 children in T_M that skipped some nodes. Such a message to v contains $l(x)$, $l(u)$, $\text{id}(u)$ and $w(\{u, v\})$. Upon receiving the call from u, v contacts its child node y in T_M whose label is a prefix of $l(x)$, forwarding u's identifier, $l(x)$ and the (updated) weight $w(\{y, u\}) = w(\{u, v\}) - w(\{v, y\})$. y then adds the edge $\{y, u\}$ with weight $w(\{y, u\})$ to the set E_D by informing u about it. Then, y continues the recursion at its child in T_M that lies in x's direction until the process reaches x itself. Since the height of T_M is $\mathcal{O}(\log n)$, u learns at most $\mathcal{O}(\log n)$ additional edges and its degree in D_T therefore is at most $\mathcal{O}(\log n)$.

Note that since the process from u propagates down the tree level by level, we can perform the algorithm at all nodes in parallel, whereby the separate construction processes follow each other in a pipelined fashion without causing too much communication. Together with Theorem 5.25, we obtain the following lemma.

Lemma 5.26 (Distance Graph). *The distance graph $D_T = (V, E_D)$ for M can be computed in time $\mathcal{O}(\log^2 n)$.*

From the way we construct the node's additional edges in E_D, and the fact that the edges in E_T preserve distances in M, we conclude the following lemma.

Lemma 5.27. *For any edge $\{u, v\} \in E_D$ it holds $w(\{u, v\}) = d_M(u, v)$.*

The next lemma is crucial for the correctness of the algorithms that follow.

Lemma 5.28 (Shortest Paths in D_T). *For every $u, v \in V$ we have that $d_{D_T}(u, v) = d_M(u, v)$ and $\mathsf{SPD}(D_T) = \mathcal{O}(\log n)$.*

Proof. To prove the lemma, we show that (1) every path from u to v in D_T has length at least $d_M(u, v)$, and (2) for every $u, v \in V$ there exists a path P from u to v in D_T with $w(P) = d_M(u, v)$ and $|P| = \mathcal{O}(\log n)$ that only contains nodes of the unique path from u to v in T_M. For (1), consider any path P from u to v in D_T. By construction of our edges, we have that $w(\{x, y\}) = d_M(x, y)$ for every edge $e \in E_D$. Therefore, by triangle inequality, we have that $w(P) \geq d_M(u, v)$.

For (2), let $P_T = (u = x_0, x_1, x_2, \dots, x_m = v)$ be the (unique) path from u to v in T_M. By the construction of T_M, $|P_T| = \mathcal{O}(\log n)$. In case that $w(P_T) = d_M(u, v)$, we are done, so let us assume that $w(P_T) > d_M(u, v)$. We show that we can replace subpaths of P_T by single edges out of $E_D \setminus E_T$ until we arrive at a path that has the desired properties.

Let w be the node in P_T that has smallest depth in T_M, which is the lowest common ancestor of u and v in T_M. We follow the *right* subpath $P_r = (w = x_i, \dots, x_m = v)$ of P_T from w to v (the *left* subpath P_ℓ from u to w is analogous). Starting at w, we sum the weights of the edges of T_M on P_r until we reach a node x_j such that the sum is higher than $d_M(w, x_j)$. In this case, the edge $\{x_{j-2}, x_{j-1}\}$ must have skipped the node x_j, i.e., x_j lies on the unique path from x_{j-2} to x_{j-1} in M. Continuing at x_j, we now follow P_r as long as we only move in the direction of x_{j-2} in M, i.e., we move to the next node if that node is closer to x_{j-2} in M than the previous one, until we stop at a node x_k. By the definition of our algorithm, there must be an edge $\{x_{j-2}, x_k\} \in E_D$ with $w(\{x_{j-2}, x_k\}) = d_M(\{x_{j-2}, x_k\})$. We replace the subpath of P_r from x_{j-2} to x_k by this edge, after which the length of the subpath of P_r from w to x_k equals $d_M(x_i, x_k)$. We continue the process starting at x_k until we reach x_m, and obtain that $w(P_r) = d_M(x_i, x_k)$. After we have performed the same process at P_ℓ (in the other direction), we have that $w(P_T) = d_M(u, v)$. Finally, note that $|P_T| = \mathcal{O}(\log n)$, so P_T has all the desired properties of the path P from the statement of the lemma. $\square$

For any node $v \in V$, we define the *nearest shortcut node* of v as $\sigma(v) := \operatorname{argmin}_{u \in \Sigma} d(v, u)$. To let each node v determine $\sigma(v)$ and $d(v, \sigma(v))$, we perform a distributed version of the Bellman-Ford algorithm that works as follows. In the first round, every shortcut node sends a message associated with its own identifier and distance value 0 to itself. In every subsequent round, every node $v \in V$ chooses the message with smallest distance value d received so far (breaking ties by choosing the one associated with the node with highest identifier), and sends a message containing $d + w(\{v, u\})$ to each neighbor u in D_T. After $\mathcal{O}(\log n)$ rounds, every node v knows the distance $d_M(v, u)$ to its closest shortcut node u in M, which we formally prove below. Since for any closest shortcut node w in G there must be a shortest path from v to w that only contains edges of M, this implies that u must also be closest to v in G, i.e., $u = \sigma(v)$, and $d_M(v, u) = d(v, \sigma(v))$.

Note that each node has only created additional edges to its descendants in T_M during the construction of D_T, therefore the degree of D_T is $\mathcal{O}(\log n)$ and we can easily perform the algorithm described above using the global network.

Lemma 5.29 (Nearest Shortcut Nodes)**.** *After $\mathcal{O}(\log n)$ rounds, each node $v \in V$ knows $id(u)$ of its nearest shortcut node $\sigma(v)$ in G and its distance $d(v, \sigma(v))$ to it.*

Proof. We first show that v learns $\mathrm{id}(u)$ and $d_M(u, v)$ of its closest shortcut node u in M (if there are multiple, let u be the one with highest identifier). Due to Lemma 5.28, we know that v will never receive a message with a smaller distance value than $d_M(u, v)$. Furthermore, there is a path $P = (u = x_1, \dots, x_k = v)$ of length $\mathcal{O}(\log n)$ from u to v with $w(P) = d_M(u, v)$. We claim that a message from u traverses the whole path P until it arrives at v after $\mathcal{O}(\log n)$ rounds. Assume to the contrary that this is not the case. Then there must exist a node x_i on the path P that does not send a message with $\mathrm{id}(u)$ and distance value $d_M(u, x_{i+1})$ to x_{i+1}. This can only happen if x_i knows a shortcut node z with $d_M(z, x_i) < d_M(u, x_i)$. This implies that z is a shortcut node with $d_M(v, z) < d_M(v, u)$, contradicting the fact that u is v's nearest shortcut node. □

Finally, the following lemma, which we will use later, implies that for each node v there is at most one edge in $E_D \setminus E_T$ to an ancestor in T_M.

Lemma 5.30 (Constant Ancestor Degree). *If our algorithm creates an edge from s to t in $E_D \setminus E_T$, where s is an ancestor of t in T_M, then there is no edge $\{v, t\} \in E_D \setminus E_T$ from any node v on the (unique) path from s to t in T_M.*

Proof. Assume there exists an edge $\{s, t\} \in E_D \setminus E_T$ and let $P = (s, v_1, \ldots, v_k = t)$ be the unique path from s to t in T_M. Since s established an edge to t, the path P from s to v_1 in M must contain node t. Furthermore, all nodes $v_1, \ldots, v_k$, and potentially many other nodes, lie in that order on the subpath of P from v_1 to t. Specifically, for each node v_i, $1 \le i \le k-1$, the subpath of P from v_i to v_{i+1} does not contain any other node of P. Therefore, our algorithm will not establish any additional edge in $E_D \setminus E_T$ from v_i to any other node of P, including t. □

5.6.3. Computing APSP between Shortcut Nodes

In this section, we first describe how the shortcut nodes can compute their pairwise distances in M by using D_T. Then, we explain how the information can be used to compute all pairwise distances between shortcut nodes in G by performing matrix multiplications.

Compute Distances in M First, each node learns the total number of shortcut nodes $n_c := |\Sigma|$, and each shortcut node is assigned a unique identifier from $[n_c]$. The first part can easily achieved using Lemma 5.4. For the second part, the nodes use the strategy depicted in Figure 3.2c of Chapter 3, where all nodes of V simulate a butterfly network, and the shortcut nodes are enumerated from 0 to $n_c - 1$ by performing an aggregation and assigning intervals to the nodes.

Note that it is impossible for a shortcut node to explicitly learn all the distances to all other shortcut nodes in polylogarithmic time, since it may have to learn $\Omega(n^{1/3})$ many bits. However, if we could distribute the distances of all $\mathcal{O}(n^{2/3})$ pairs of shortcut nodes uniformly among all nodes of V, each node would only have to store $\mathcal{O}(\log n)$ bits. We make use of this in the following way. To each pair (i, j) of shortcut nodes we assign a *representative* $h(i, j) \in V$, which is chosen using a (pseudo-)random hash function $h : [n_c]^2 \to V$ that is known to all nodes and that satisfies $h(i, j) = h(j, i)$. Note that for a node $v \in V$ there can be up to $\mathcal{O}(\log n)$ keys (i, j) for which $h(i, j) = v$, w.h.p., thus v has to act on behalf of at most $\mathcal{O}(\log n)$ nodes. The goal of $h(i, j)$ is to infer $d_M(i, j)$ from learning all the edges on the shortest path from i to j in D_T.

To do so, the representative $h(i,j)$ first has to retrieve the labels $l(i)$ and $l(j)$ of i and j in T_M. However, i cannot send this information directly, as it would have to reach the representatives of every shortcut node pair (i,k), of which there may be up to $\Omega(n^{1/3})$ many. Instead, it performs a multicast using our techniques from Chapter 4 to inform all these representatives. To that end, $h(i,j)$ first joins the multicast groups of i and j by participating in the construction of multicast trees using Theorem 4.11 with parameters $\ell = \mathcal{O}(\log n)$ and $L = \mathcal{O}(n^{2/3})$ (since each node acts for at most $\mathcal{O}(\log n)$ of the $\mathcal{O}(n^{2/3})$ representatives, w.h.p., and each representative joins two multicast groups). Therefore, the multicast trees are computed in time $\mathcal{O}(\log n)$ and have congestion $C = \mathcal{O}(\log n)$, w.h.p. We then use Theorem 4.12 to let each shortcut node i multicast its label $l(i)$ to all representatives $h(i,k)$. With parameter $\hat{\ell}$ as the maximum number of representatives simulated by the same node, which can easily be computed using Lemma 5.4 on M, and congestion C, the theorem gives a runtime of $\mathcal{O}(\log n)$, w.h.p.

From the knowledge of $l(i)$ and $l(j)$, $h(i,j)$ can easily infer the labels of all nodes on the path P from i to j in T_M. Specifically, it knows the label $l(x)$ of the lowest common ancestor x of i and j in T_M, which is simply the longest common prefix of $l(i)$ and $l(j)$. We remark that, technically, $h(i,j)$ can only infer the exact labels of the nodes of P if it knew the degree of every node in T_M. To circumvent this, $h(i,j)$ simply assumes that the tree is binary, which implies that some nodes of P (apart from i, j, and x) may not actually exist. However, as this is not a problem for the algorithm, we disregard this issue in the remainder of this section.

The goal of $h(i,j)$ is to retrieve the edge from each node $v \in P \setminus \{x\}$ to its parent in T_M, as well as v's additional edge to an ancestor in D_T, of which there can be at most one by Lemma 5.30. Since by Lemma 5.28 these edges contain a shortest path from i to j that preserves the distance in M, $h(i,j)$ can easily compute $d_M(i,j)$ using this information.

To retrieve the edges, $h(i,j)$ joins the multicast groups of all nodes of $P \setminus \{x\}$ using Theorem 4.11. Then, each inner node of T_M performs a multicast using Theorem 4.12 to inform all nodes in its multicast group about the edge to its parent in T_M, and the edge of $E_D \setminus E_T$ to an ancestor, if it exists (see Lemma 5.30). Since each node acts on behalf of at most $\mathcal{O}(\log n)$ representatives, and each representative joins $\mathcal{O}(\log n)$ multicast groups, all can be done in $\mathcal{O}(\log n)$ rounds, w.h.p. Since a shortest path between i and j in D_T must be comprised of a subset of the acquired edges by the proof of Lemma 5.28, we conclude the following lemma.

Lemma 5.31. *Every representative $h(i,j)$ learns $d_M(i,j)$ in time $\mathcal{O}(\log n)$, w.h.p.*

Compute Distances in G Let $A \in \mathbb{N}_0^{n_c \times n_c}$ be the *distance matrix* of the shortcut nodes, where

$$A_{i,j} = \begin{cases} \min\{w(\{i,j\}), d_M(i,j)\} & \text{if } \{i,j\} \in E, \\ d_M(i,j) & \text{otherwise.} \end{cases}$$

Our goal is to square A for $\lceil \log n \rceil + 2$ many iterations in the *min-plus semiring*. More precisely, we define $A^1 = A$, and for $t \geq 1$ we have that

$$A^{2^t}_{i,j} = \min_{k \in [n_c]} (A^{2^{t-1}}_{i,k} + A^{2^{t-1}}_{k,j}).$$

The following lemma shows that after squaring the matrix $\lceil \log n \rceil + 2$ times, its entries contain the distances in G.

Lemma 5.32. *$A_{i,j}^{2^{\lceil \log n \rceil + 2}} = d(i,j)$ for each $i, j \in \Sigma$.*

Proof. First, note that $A_{i,j}^t \geq d(i,j)$ for all $i, j \in \Sigma$ and all $t \geq 1$, since every entry corresponds to the length of an actual path in G. We show by induction on t that for all $t \geq 2$, we have that $A_{i,j}^{2^t} \leq \min_{P \in \mathcal{P}(i,j,t-2)} w(P)$, where $\mathcal{P}(i,j,t)$ is the set of all paths from i to j in G that contain at most 2^t non-tree edges. Since any shortest path between two shortcut nodes i and j contains at most $n-1$ non-tree edges, $A_{i,j}^{2^{\lceil \log n \rceil + 2}} = d(i,j)$.

To establish the base case, we first show that for $t = 2$, $A_{i,j}^4 \leq \min_{P \in \mathcal{P}(i,j,0)} w(P)$. Let P be a path from i to j that contains at most $2^0 = 1$ non-tree edge $\{v_1, v_2\}$ such that $w(P) = \min_{P' \in \mathcal{P}(i,j,0)} w(P')$. W.l.o.g., assume that v_1 appears first in P (v_1 might be i). Let P_1 be the subpath of P from i to v_1, and P_2 be the subpath from v_2 to j. Note that $A_{i,v_1}^1 \leq w(P_1)$, $A_{v_1,v_2}^1 \leq w(e)$, and $A_{v_2,j}^1 \leq w(P_2)$. Therefore, $A_{i,v_2}^2 \leq w(P_1) + w(e)$ and $A_{v_2,j}^2 \leq w(P_2)$, which implies that

$$A_{i,j}^4 \leq A_{i,v_2}^2 + A_{v_2,j}^2 \leq w(P_1) + w(e) + w(P_2) = w(P) = \min_{P' \in \mathcal{P}(i,j,0)} w(P').$$

Now let $t > 2$ and consider a path $P \in \mathcal{P}(i,j,t-2)$ such that $w(P) = \min_{P \in \mathcal{P}(i,j,t-2)}$. Since P contains at most $2^{t-2} \geq 2$ non-tree edges, we can divide P at some shortcut node $k \in \Sigma$ into two paths P_1 and P_2 that both contain at most 2^{t-3} non-tree edges. We have that

$$\begin{aligned}
A_{i,j}^{2^t} &\leq A_{i,k}^{2^{t-1}} + A_{i,k}^{2^{t-1}} \\
&\leq \min_{P' \in \mathcal{P}(i,k,t-3)} w(P') + \min_{P' \in \mathcal{P}(k,j,t-3)} w(P') \\
&\leq w(P_1) + w(P_2) \\
&= w(P) = \min_{P' \in \mathcal{P}(i,j,t-2)} w(P'),
\end{aligned}$$

which concludes the proof. □

We now describe how the matrix can efficiently be multiplied. As an invariant to our algorithm, we show that at the beginning of the t-th multiplication, every representative $h(i,j)$ stores $A_{i,j}^{2^{t-1}}$. Thus, for the induction basis we first need to ensure that every representative $h(i,j)$ learns $A_{i,j}$. By Lemma 5.31, $h(i,j)$ already knows $d_M(i,j)$, thus it only needs to retrieve $w(\{i,j\})$, if that edge exists. To do so, we first compute an orientation with outdegree $\mathcal{O}(\log n)$ in time $\mathcal{O}(\log n)$ using [BE10, Corollary 3.12] in the local network. For every edge $\{i,j\}$ that is directed from i to j, i sends a message containing $w(\{i,j\})$ to $h(i,j)$; since the arboricity of G is $\mathcal{O}(\log n)$, every node only has to send at most $\mathcal{O}(\log n)$ messages.

The t-th multiplication is then done in the following way. We use a (pseudo-)random hash function $h : [n_c]^3 \to V$, where $h(i,j,k) = h(j,i,k)$. First, every node $h(i,j,k) \in V$ needs to learn $A_{i,j}^{2^{t-1}}$ (we will again ignore the fact that a node may have to act on behalf of at most $\mathcal{O}(\log n)$ nodes $h(i,j,k)$). To do so, $h(i,j,k)$ joins the multicast group of $h(i,j)$ using Theorem 4.11. By performing a multicast using Theorem 4.12, $h(i,j)$ can then send $A_{i,j}^{t-1}$ to all $h(i,j,k)$. Since there are $L \leq [n_c]^3 = \mathcal{O}(n)$ nodes $h(i,j,k)$ that each join a multicast group, and each node needs to send and receive at most $\ell = \mathcal{O}(\log n)$ values, w.h.p., the theorems imply a runtime of $\mathcal{O}(\log n)$, w.h.p.

After $h(i,j,k)$ has received $A_{i,j}^{2^{t-1}}$, it sends it to both $h(i,k,j)$ and $h(k,j,i)$. It is easy to see that thereby $h(i,j,k)$ will receive $A_{i,k}^{2^{t-1}}$ from $h(i,k,j)$ and $A_{k,j}^{2^{t-1}}$ from $h(k,j,i)$. Afterwards, $h(i,j,k)$ sends the value $A_{i,k}^{2^{t-1}} + A_{k,j}^{2^{t-1}}$ to $h(i,j)$ by participating in an aggregation using Theorem 4.3 and the minimum function, whereby $h(i,j)$ receives $A_{i,j}^{2t}$. By the same arguments as before, $L = \mathcal{O}(n)$, and $\ell = \mathcal{O}(\log n)$, which implies a runtime of $\mathcal{O}(\log n)$, w.h.p. We conclude the following lemma.

Lemma 5.33 (Shortcut Nodes Distances)**.** *After $\lceil \log n \rceil + 2$ many matrix multiplications, $h(i,j)$ stores $d(i,j)$ for every $i,j \in [n_c]$. The total number of rounds is $\mathcal{O}(\log^2 n)$, w.h.p.*

5.6.4. Approximating SSSP and the Diameter

We are now all set in order to compute approximate distances between any two nodes $s,t \in V$. Specifically, we approximate $d(s,t)$ by

$$\widetilde{d}(s,t) = \min\{d_M(s,t),\, d(s,\sigma(s)) + d(\sigma(s),\sigma(t)) + d(\sigma(t),t)\}.$$

We first show that $\widetilde{d}(s,t)$ gives a 3-approximation for $d(s,t)$.

Lemma 5.34 (3-Approximation)**.** *For any $s,t \in V$ it holds that*

$$d(s,t) \leq \widetilde{d}(s,t) \leq 3 \cdot d(s,t).$$

Proof. Obviously, $\widetilde{d}(s,t)$ represents the length of a path from s to t, so $d(s,t) \leq \widetilde{d}(s,t)$ holds. If there exists a shortest path between s and t that does not contain any shortcut node, then $\widetilde{d}(s,t) \leq d_M(s,t) = d(s,t)$. Therefore, assume that any shortest path P from s to t contains at least one non-tree edge. Choose $x, y \in \Sigma$ such that $d(s,t) = d(s,x) + d(x,y) + d(y,t)$ (see Figure 5.7 for an illustration).

Since $\sigma(s)$ is the nearest shortcut node of s, we have $d(s,\sigma(s)) \leq d(s,x)$ and, analogously, $d(\sigma(t),t) \leq d(y,t)$. Also, by the triangle inequality, we have that

$$\begin{aligned} d(\sigma(s),\sigma(t)) &\leq d(\sigma(s),s) + d(s,x) + d(x,y) + d(y,t) + d(t,\sigma(t)) \\ &= d(s,\sigma(s)) + d(s,t) + d(t,\sigma(t)). \end{aligned}$$

Putting all pieces together, we get

$$\begin{aligned} \widetilde{d}(s,t) &\leq d(s,\sigma(s)) + d(\sigma(s),\sigma(t)) + d(\sigma(t),t) \\ &\leq d(s,\sigma(s)) + d(s,\sigma(s)) + d(s,t) + d(t,\sigma(t)) + d(\sigma(t),t) \\ &= 2(d(s,\sigma(s)) + d(\sigma(t),t)) + d(s,t) \\ &\leq 2(d(s,x) + d(y,t)) + d(s,t) \\ &\leq 2d(s,t) + d(s,t) \\ &= 3d(s,t). \end{aligned}$$

□

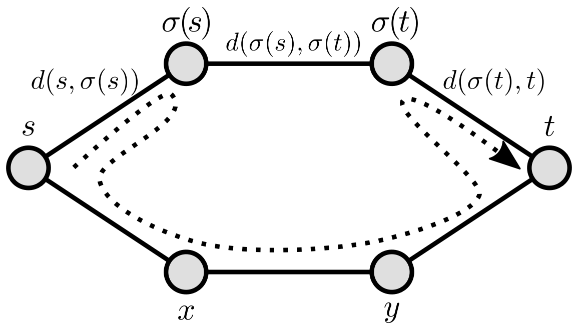


Figure 5.7.: Illustration for the computation of the approximate distance between nodes s and t. We have that $d(s,t) = d(s,x) + d(x,y) + d(y,t)$. By the triangle inequality, the length $d(s,\sigma(s)) + d(\sigma(s),\sigma(t)) + d(\sigma(t),t)$ is at most the length of the dotted path, which, since $d(s,\sigma(s)) \leq d(s,x)$ and $d(\sigma(t),t) \leq d(y,t)$ has length at most $3d(s,t)$.

SSSP Approximation To approximate SSSP, every node v needs to learn $\widetilde{d}(s,v)$ for a given source s. To do so, the nodes first have to compute $d_M(s,v)$, which can be done in time $\mathcal{O}(\log n)$ by performing SSSP in M using Theorem 5.6. Then, the nodes construct D_T in time $\mathcal{O}(\log^2 n)$ using Lemma 5.26. With the help of D_T and Lemma 5.29, s can compute $d(s,\sigma(s))$, which is then broadcast to all nodes in time $\mathcal{O}(\log n)$ using Lemma 5.4. Then, we compute all pairwise distances in G between all shortcut nodes in time $\mathcal{O}(\log^2 n)$, w.h.p., using Lemma 5.33; specifically, every shortcut node v learns $d(\sigma(s),v)$. By performing a slight variant of the algorithm of Lemma 5.29, we can make sure that every node t not only learns its closest shortcut node $\sigma(t)$ in M, but also retrieves $d(\sigma(s),\sigma(t))$ from $\sigma(t)$ within $\mathcal{O}(\log n)$ rounds. Since t is now able to compute $\widetilde{d}(s,t)$, we conclude the following theorem.

Theorem 5.35 (Sparse Graphs SSSP)**.** *3-approximate SSSP can be computed in graphs that contain at most $n + \mathcal{O}(n^{1/3})$ edges and have arboricity $\mathcal{O}(\log n)$ in time $\mathcal{O}(\log^2 n)$, w.h.p.*

Diameter Approximation For a 3-approximation of the diameter, consider

$$\widetilde{D} = 2 \max_{s \in V} d(s,\sigma(s)) + \max_{x,y \in \Sigma} d(x,y).$$

$\widetilde{D}$ can be computed using Lemmas 5.26, 5.29, and 5.33, and by using Lemma 5.4 on M to determine the maxima of the obtained values. By the triangle inequality, we have that $D \leq \widetilde{D}$. Furthermore, since $d(s,\sigma(s)) \leq D$ and $\max_{x,y\in\Sigma} d(x,y) \leq D$, we have that $\widetilde{D} \leq 3D$. We conclude the final theorem of this chapter.

Theorem 5.36 (Sparse Graphs Diameter)**.** *A 3-approximation of the diameter can be computed in graphs that contain at most $n + \mathcal{O}(n^{1/3})$ edges and have arboricity $\mathcal{O}(\log n)$ in time $\mathcal{O}(\log^2 n)$, w.h.p.*

5.7. Outlook

The results in this chapter indicate that some shortest path problems can be computed very efficiently in sparse graphs. It may certainly be interesting to extend our results to more general graph classes such as *outerplanar* or *planar graphs*, or to extend the number of edges that the algorithm of Section 5.6 can handle. We believe that some of the tools and techniques developed in this chapter may be helpful towards that goal, for example, our redistribution framework to use the Euler tour technique and other algorithms in high-degree graphs, our extension of the result of Götte et al. [Göt+20] to compute spanning trees in the NCC_0, or our matrix multiplication algorithm. Potentially in combination with sparse spanner constructions (see, e.g., [BS07]) or skeletons (e.g., [UY91]), our algorithms may lead to efficient shortest path algorithms in more general graph classes. Also, our algorithm to construct a hierarchical tree decomposition may be of independent interest, as such constructions are used for example in routing algorithms for wireless networks (see, e.g., [GZ05; Kap+18]).

6 Shortest Paths in General Hybrid Networks

After having investigated shortest path problems in sparse graphs, we conclude our study of hybrid networks by considering *general* graphs. More precisely, we assume that the local network G can form *any* graph, and develop efficient algorithms for the Single-Source Shortest Paths (SSSP) Problem.

Since the algorithms of Chapter 5 crucially rely on the sparsity of the underlying graph, a straightforward extension of our results to general graphs is, unfortunately, futile. Although there does not exist a formal lower bound for SSSP, the $\widetilde{\Omega}(\sqrt{n})$ lower bound for APSP [Aug+20b] and the $\widetilde{\Omega}(n^{1/3})$ lower bound for the Diameter Problem [KS20] also suggest the difficulty of this problem in general graphs. To cope with the additional complexity imposed by dense graphs, we allow the nodes to perform an arbitrary amount of local communication. Formally, we shift our attention to the $\mathsf{LOCAL}+\mathsf{NCC}$ model, for which we have $\lambda = \infty$ and $\gamma = \mathcal{O}(\log n)$. Note that the above-mentioned lower bounds also hold in this model.

From a theoretical perspective, the $\mathsf{LOCAL}+\mathsf{NCC}$ model, as a combination of the most permissive LOCAL model for local edges and the very restrictive node-capacitated clique model for the global edges, is particularly clean and well-suited to investigate the power of hybrid networks. Moreover, we believe that the practical relevance of this model is justified by the fact that direct connections between devices are typically highly efficient and offer a large bandwidth at comparatively low cost, whereas communication over a shared global communication network such as the Internet, satellites, or the cellular network, is costly and typically offers only a comparatively small data rate.

This chapter contains two randomized algorithms for SSSP, an exact algorithm with a runtime of $\widetilde{\mathcal{O}}(\sqrt{\mathsf{SPD}})$, and an algorithm that computes $(1/\varepsilon)^{\mathcal{O}(1/\varepsilon)}$-approximate SSSP in time $\widetilde{\mathcal{O}}(n^{\varepsilon})$, w.h.p. We demonstrate that by making use of both local and global communication, we can achieve significant runtime improvements for this class of problems compared to using local or global edges alone, which highlights the importance of exploiting hybrid communication capabilities of modern networks.

Underlying Publication The chapter is based on the following publication.

> J. Augustine, K. Hinnenthal, F. Kuhn, C. Scheideler, and P. Schneider. "Shortest Paths in a Hybrid Network Model". In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2020, pp. 1280–1299 [Aug+20b]

We focus on the second part of the paper, more precisely, on the exact and the $\widetilde{\mathcal{O}}(n^{\varepsilon})$-time approximate SSSP algorithms. An overview of the remaining results of the publication can be found below.

Outline The introductory section of this chapter contains an overview of our results in comparison to related and subsequent work. The technical part is divided into the two main results of this chapter. In Section 6.1, we describe an algorithm for exact SSSP. The algorithm is complemented by an approximation algorithm for SSSP, which is described in Section 6.2. Both sections first contain a concise description of each algorithm, followed by a detailed description and analysis. An outlook on future work concludes the hybrid network part of this dissertation.

Contribution We present two randomized algorithms for the SSSP Problem in the $\mathsf{LOCAL}+\mathsf{NCC}$ model. In Section 6.1, we show that the SSSP Problem can be solved exactly in time $\widetilde{\mathcal{O}}(\sqrt{\mathsf{SPD}})$, w.h.p. Recall that SPD denotes the *shortest-path diameter*, which is the smallest value h such that there exists a shortest path with h many hops between any pair of nodes in G. Note that $\mathsf{SPD} < n$, since edge weights are non-negative. The algorithm combines the local and global network in the following way. Using the local network, every node learns the graph up to a distance of $2\sqrt{\mathsf{SPD}}$ hops, thus learning the $\sqrt{\mathsf{SPD}}$ neighborhood of any node within $\sqrt{\mathsf{SPD}}$ hops. This knowledge is used to distribute distance information from the source s in an iterative fashion over the global network, where in iteration i, all nodes that have a shortest path to s with $\mathcal{O}(i^2)$ hops learn their distance to s. An iteration takes only $\widetilde{\mathcal{O}}(1)$ rounds, leveraging a divide-and-conquer approach to distribute the information in each node's neighborhood, and the Aggregation Algorithm of Chapter 4. Note that the best known algorithm that uses *only* global edges requires $\Omega(\mathsf{SPD})$ rounds (see Section 4.4.2 of Chapter 4). For the local network (without global communication), we have an obvious lower bound of $\Omega(\mathfrak{D})$, which, since $\mathfrak{D} = \mathsf{SPD}$ in unweighted graphs, also implies a lower bound of $\Omega(\mathsf{SPD})$. By combining local and global communication, we obtain an algorithm that substantially improves upon this runtime.

Our second algorithm, which we describe in Section 6.2, provides an approximate solution for SSSP. The algorithm is based on recursively building a hierarchy of $\mathcal{O}(\log_{\alpha} n)$ *skeleton spanners*, which are spanners of *skeleton graphs*, for some $\alpha > 1$. Roughly speaking, given some skeleton spanner H, we obtain the next coarser skeleton spanner H' by sampling each node of H with probability $1/\alpha$ and computing a spanner with a good stretch on the sampled nodes. As a technical result, we show that given a low arboricity graph H, we can efficiently compute a low arboricity spanner H' of H using only global edges. By choosing $\alpha = n^{\varepsilon/3}$ for some $\varepsilon > 0$, we show the hierarchy of skeleton spanners can be used to compute $(1/\varepsilon)^{\mathcal{O}(1/\varepsilon)}$-approximate SSSP in time $\widetilde{\mathcal{O}}(n^{\varepsilon})$, w.h.p. For any constant $\varepsilon > 0$, we get a constant SSSP approximation, albeit with a potentially large constant. Choosing $(1/\varepsilon) = \sqrt{\log n / \log\log n}$ to balance time and approximation factor, the algorithm computes a subpolynomial $2^{\mathcal{O}(\sqrt{\log n \log\log n})}$-approximate SSSP solution in time $\widetilde{\mathcal{O}}(2^{\sqrt{\log n \log\log n}})$.

Further Results of the Publication In this chapter, we mainly focus on the second part of the underlying publication [Aug+20b], which revolves around the

SSSP Problem. The first part of the publication, which we omit, is mostly concerned with the APSP Problem. However, to solve APSP, we employ a set of techniques that is substantially different from what we present in this chapter. Since it is of independent interest, however, we briefly describe the main results in this section.

All results of the publication omitted in this chapter are based on an algorithm to solve the *token dissemination* problem. The goal of this problem is to broadcast a set of tokens of size $\mathcal{O}(\log n)$, each of which is initially only known by one node. The main idea behind the algorithm is to duplicate the tokens, and then randomly disseminate them via global edges. By choosing the parameters of the algorithm appropriately, the algorithm ensures that a copy of each token lies within a relatively small neighborhood of each node, and can thus be collected in the local network within few rounds. Specifically, if k is the number of distinct tokens and ℓ is the initial maximum number of tokens per node, the algorithm solves the token dissemination problem within $\widetilde{\mathcal{O}}(\sqrt{k} + \ell)$ rounds, w.h.p.

The main contribution of the first part of the publication is an APSP algorithm that combines the token dissemination algorithm with the classical approach of building *skeleton graphs* [UY91]. The basic idea is to sample a set of nodes with some probability $1/x$ and then compute virtual edges among pairs of sampled nodes connected by a path of at most $h = \widetilde{\mathcal{O}}(x)$ hops in G. Using the token dissemination algorithm, all virtual edges can be made publicly known. Furthermore, each node can broadcast the distance to its closest skeleton node, which must lie within h hops, w.h.p. With the global knowledge gained in this way, each node can compute its distance to any other node with sufficient hop distance; shorter distances can easily be handled using the local network instead. The technique has, for example, been used by Forster and Nanongkai in the CONGEST model [FN18, Lemma 5] and Dory and Parter in the congested clique [DP20, Lemma 8].

By balancing the parameter x to optimize the overall runtime, the approach leads to an exact APSP algorithm with a runtime of $\widetilde{\mathcal{O}}(n^{2/3})$, w.h.p. Additionally, we present a 3-approximate APSP algorithm with runtime $\widetilde{\mathcal{O}}(\sqrt{n})$ for weighted graphs, and a $(1+\varepsilon)$-approximate APSP algorithm with runtime $\widetilde{\mathcal{O}}(\sqrt{n/\varepsilon})$ that rely on the same approach. Note that this is significantly faster than the $\widetilde{\Omega}(n)$ bound if only either the local or the global network could be used. These bounds immediately follow from the facts that the diameter of the local network might be $\Omega(n)$ and that every node can only receive $\mathcal{O}(\log n)$ messages over global edges. The APSP upper bounds are complemented by a lower bound that states that even computing an α-approximate solution for some $\alpha = \widetilde{\mathcal{O}}(\sqrt{n})$ requires at least $\widetilde{\Omega}(\sqrt{n})$ rounds. Therefore, the approximate APSP algorithms contained in our publication are tight up to polylogarithmic factors.

The combination of skeleton graphs and the token dissemination approach also allows to efficiently simulate algorithms for the *broadcast congested clique* model (BCC). The BCC is a weaker variant of the congested clique, in which nodes can only send the *same* $\mathcal{O}(\log n)$-bit message to all other nodes in each round, but may receive an arbitrary number of distinct messages. More precisely, using token dissemination a single round of the BCC in a skeleton that contains $\widetilde{\mathcal{O}}(n^{2/3})$ nodes can be performed in time $\widetilde{\mathcal{O}}(n^{1/3})$, w.h.p. By using a simulation of the $\widetilde{\mathcal{O}}(1)$-time algorithm for $(1+\varepsilon)$-approximate SSSP in the BCC by Becker et al. [Bec+17], we can therefore compute approximate distances in the skeleton in time $\widetilde{\mathcal{O}}(n^{1/3})$, w.h.p. For every other node, it suffices to learn the h-limited distance to all skeleton nodes within h hops for $h = \widetilde{\mathcal{O}}(n^{1/3})$ as well as their distance to s in the skeleton, which can be done in time $\widetilde{\mathcal{O}}(n^{1/3})$ as well. All details of the APSP algorithms, as well as the $(1+\varepsilon)$-approximate SSSP algorithm, can be found in the underlying publication of this chapter [Aug+20b].

Related and Subsequent Work To the best of our knowledge, this chapter's original publication [Aug+20b] contains the first rigorous algorithms for shortest paths in hybrid networks. Since then, several papers for our hybrid model have been published, some of which considering shortest path problems as well [FHS20; KS20; CLP20; Göt+20]. All of these papers have been discussed already in this thesis, mainly in Chapter 5. Therefore, we focus on the results that are most closely related to this chapter, and give a more extensive overview of the results of Kuhn and Schneider [KS20] and Censor-Hillel et al. [CLP20]. For an overview of shortest path results in the CONGEST model, specifically the congested clique, and in PRAMs, we refer the reader to Chapters 4 and 5.

Schneider and Kuhn [KS20] were the first to improve upon the results of this chapter's publication. First, the $\widetilde{\mathcal{O}}(n^{2/3})$-time algorithm for exact APSP was improved to achieve a runtime of $\widetilde{\mathcal{O}}(\sqrt{n})$, w.h.p., matching the lower bound up to polylogarithmic factors. At the heart of the algorithm lies an efficient algorithm for *token routing problems*, which allows to perform the necessary communication between the nodes more efficiently than the token dissemination algorithm described above. More precisely, the APSP algorithm leverages the fact that the nodes do not actually need to learn *all* pairwise distances, but only their *own* distance to all other nodes. Using the token routing algorithm, this information can be concentrated on the nodes of a skeleton of size $\widetilde{\Omega}(\sqrt{n})$, who subsequently inform all nodes within their local $\widetilde{\mathcal{O}}(\sqrt{n})$-hop neighborhood about their APSP information. From this information, the nodes can easily infer their distances to all other nodes.

Using the token routing scheme, Kuhn and Schneider further provide an efficient simulation scheme for congested clique algorithms. Using this framework, they derive constant approximations for the *k-Source Shortest Paths (k-SSP) Problem* with runtime $\widetilde{\mathcal{O}}(\sqrt{k})$, which is optimal up to polylogarithmic factors for large k. For SSSP, they present an exact $\widetilde{\mathcal{O}}(n^{2/5})$-time algorithm. Additionally, they derive algorithms to approximate the hop-diameter $\mathfrak{D}$ with runtimes $\widetilde{\mathcal{O}}(n^{1/3}/\varepsilon)$ and $\widetilde{\mathcal{O}}(n^{0.397}/\varepsilon)$ and approximation factors $(3/2+\varepsilon)$ and $(1+\varepsilon)$, respectively. The upper bounds prove that the hop-diameter can be computed more efficiently than APSP, at least in the approximate case, and are complemented by a lower bound that shows that computing the exact diameter takes $\widetilde{\Omega}(n^{1/3})$ rounds.

Recently, Censor-Hillel et al. [CLP20] improved upon various bounds for distance computations in the hybrid model (a comprehensive comparison of their contribution to previous results is provided in their paper). Notably, they present an algorithm for exact SSSP in time $\widetilde{\mathcal{O}}(n^{1/3})$, w.h.p., which is faster than our $\widetilde{\mathcal{O}}(\sqrt{\mathsf{SPD}})$-time algorithm if $\mathsf{SPD} \geq n^{2/3}$ is comparably large. The algorithm combines the skeleton approach described above with the idea to simulate an *Oracle*, which is a single node that essentially learns the entire skeleton graph. The Oracle is the node that has highest degree, which allows all other nodes to distribute information about their incident edges among the Oracle's neighbors using a congested clique simulation. Using the local network, the Oracle can then collect all information.

The Oracle simulation approach is then extended to *Tiered Oracles*, in which each node u in a skeleton S learns the messages of all nodes whose degree in S is at most twice the degree of u. Using this refined approach, the authors present a multitude of improvements upon previous results. For instance, they show that the $\widetilde{\mathcal{O}}(n^{1/3})$-time algorithm for SSSP can be extended to $n^{1/3}$ sources. Additionally, they derive an $\widetilde{\mathcal{O}}(n^{1/3}/\varepsilon + n^{x/2})$-time algorithm for unweighted $(1+\varepsilon)$-approximate n^x-SSP; for weighted graphs, a 3-approximate solution can be found in time $\widetilde{\mathcal{O}}(n^{1/3}+n^{x/2})$, w.h.p. Among other results, they also improve upon the hop-diameter results of Kuhn and Schneider [KS20], proving an $\widetilde{\mathcal{O}}(n^{1/3}/\varepsilon)$ time bound for a $(1+\varepsilon)$-approximation.

As noted before in Chapter 5, we can approximate SSSP by simulating the $(1+\varepsilon)$-approximate polylogarithmic-time algorithm for PRAMs by Li [Li20]. To do so, we first compute a spanner of G, e.g., by using the spanner algorithm of Baswana and Sen [BS07], and then simulate the PRAM algorithm on the spanner. This way, we can for example compute constant approximations for SSSP in time $\widetilde{\mathcal{O}}(n^{\varepsilon})$, w.h.p., which gives an alternative to the approximation algorithm presented in this chapter. By first computing an $\mathcal{O}(\log n)$-spanner with arboricity $\widetilde{\mathcal{O}}(1)$, we can also compute $\mathcal{O}(\log n)$-approximate SSSP in time $\widetilde{\mathcal{O}}(1)$, w.h.p. However, the algorithm of Li is very complicated and, compared to our approach, hides huge polylogarithmic terms in its runtime.

6.1. Algorithm for Exact SSSP

Before we give a detailed account of our exact SSSP algorithm in Section 6.1.2, in the following section we first provide a rather intuitive explanation and state our main result.

6.1.1. Overview

Throughout the algorithm, every node v maintains a distance estimate $\hat{d}(v)$, which is initialized to $\hat{d}(v) = \infty$ for $v \in V \setminus \{s\}$, and $\hat{d}(s) = 0$. Eventually, we will have that $\hat{d}(v) = d(s,v)$ for every node $v \in V$.

For each node $v \in V$, let $V(v,i) := \{u \in V \mid \text{hop}(v,u) \leq i\}$ and let $G(v,i)$ be the subgraph of G induced by all nodes of $V(v,i)$. Furthermore, let $t(i) := \sum_{j=1}^{i} j$ denote the i-th triangular number. The algorithm proceeds in phases $i = 1, \dots, \lceil\sqrt{2\,\mathsf{SPD}}\rceil$, and we maintain the following two invariants for each phase.

Proposition 6.1 (Phase Invariants)**.** *At the beginning of each phase i, we have that*

(1) every node $v \in V$ knows $G(v, 2i-2)$,

(2) $\hat{d}(v) = d(s,v)$ for every $v \in V$ with $sph(s,v) \leq t(i-1)$.

Note that Invariant 2 *only* implies that nodes within a shortest path hop-distance of at most $t(i-1)$ from s know the correct distance to s; all other nodes may have learned distance estimates that are far off. However, since there exists a shortest path of at most SPD hops between any two nodes, and since

$$t(\lceil\sqrt{2\,\mathsf{SPD}}\,\rceil) = \lceil\sqrt{2\,\mathsf{SPD}}\,\rceil(\lceil\sqrt{2\,\mathsf{SPD}}\,\rceil + 1)/2 \geq \mathsf{SPD},$$

after $\lceil\sqrt{2\,\mathsf{SPD}}\,\rceil$ phases every node knows its exact distance to s. We will later show that each phase only takes time $\widetilde{\mathcal{O}}(1)$, w.h.p., which implies a total runtime of $\widetilde{\mathcal{O}}(\sqrt{\mathsf{SPD}})$.

Maintaining Invariant (1) is simple: Every node sends all information it has learned about the graph so far to its neighbors for two rounds via local edges at the beginning of each phase. Maintaining Invariant (2) is the main concern of our algorithm.

If $\text{sph}(s,v) \leq t(i) = i + t(i-1)$ for some node $v \in V$, then there must be a node on a shortest path from s to v that is within i hops from v (i.e., a node of $G(v,i)$) and that knows its correct distance to s already by Invariant (2). Since v knows $d_{G(v,i)}(v,u) = d_{G(u,i)}(u,v)$ for every $u \in G(v,i)$ after Invariant (1) has been established for the next phase, it would suffice for v to learn the distance estimate of all nodes in $G(v,i)$, and could then determine $d(s,v)$ with the equation

$$d(s,v) = \min_{u \in G(v,i)} \Big(\hat{d}(u) + d_{G(u,i)}(u,v)\Big). \tag{6.1}$$

Unfortunately, naively exchanging all distance estimates among all pairs of nodes within i hops of each other over the global network in order to compute Equation 6.1 is not possible in time $\widetilde{\mathcal{O}}(1)$, as $G(v,i)$ could be of size $\Theta(n)$. However, we will exploit the fact that it suffices that node v learns the distance label

$$d_{uv} := \hat{d}(u) + d_{G(u,i)}(u,v)$$

from some node u that minimizes Equation 6.1 and can safely disregard distance labels of other nodes in $G(v,i)$.

Propagate Distance Labels We define $T(u,i)$ as the shortest-path tree of $G(u,i)$ rooted at u, in which the parent of every node v is a neighbor w in $G(u,i)$ such that $d_{G(u,i)}(u,v) = d_{G(u,i)}(u,w) + w(\{u,v\})$. If there are multiple such neighbors, we choose the one that minimizes $\text{sph}_{G(u,i)}(u,w)$, breaking ties by choosing the node w with smallest identifier. Note that of all shortest paths from u to v in $G(u,i)$, $T(u,i)$ contains a shortest path with fewest hops.

Due to Invariant (1), u clearly knows $T(u,i)$. Furthermore, since it knows all nodes within hop-distance $2i$, it also knows $T(v,i)$ for every $v \in V(u,i)$, and analogously, every node $v \in V(u,i)$ knows $T(u,i)$. The goal of u is to propagate the distance label d_{uv} to every node v in $T(u,i)$ for which u minimizes Equation 6.1. To achieve that, u initiates a recursive divide-and-conquer strategy, where the executions of all nodes are performed in parallel. An example of the process can be found in Figure 6.1. First, u starts with the tree $T := T(u,i)$. In each recursion level, T is divided at a *splitting node* σ whose removal decomposes T into subtrees of size at most $|V[T]|/2$ (see Figure 6.1a). As we show in Lemma 6.3, such a node can easily be found deterministically; moreover, since every node $v \in V(u,i)$ knows $T(u,i)$, v also knows the splitting node of each level of u's recursion.

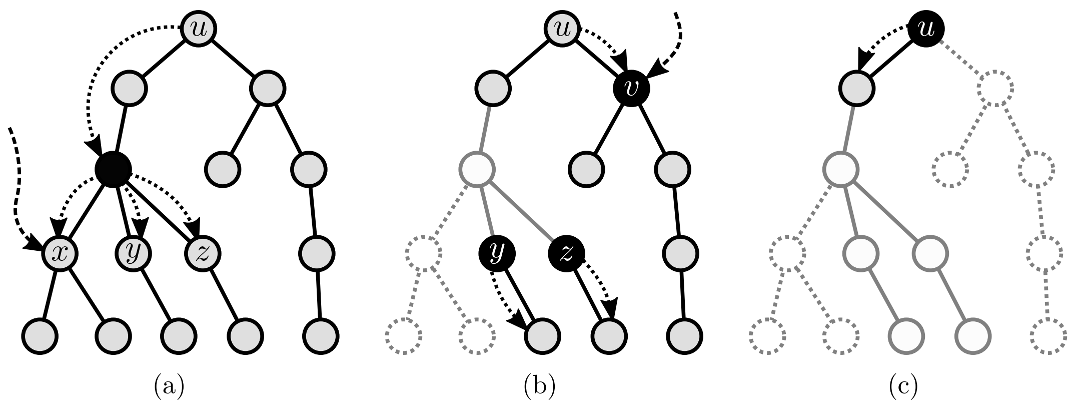


Figure 6.1.: An example of the propagation of distance information within the shortest-path tree $T(u,i)$ of a node u. (a) In the first step, u selects the black node as its splitting node, which subsequently forwards distance information to its children in $T(u,i)$. Child x receives better distance information by some node and will not continue the recursion at its subtree; for y and z, however, u's distance value is still the best, and they continue the recursion. (b) Recursions are continued in three different components of $T(u,i)$, one that is rooted at u, and two that are rooted at y and z. y and z choose themselves as splitting nodes, whereas u chooses its child v. However, v receives a better distance value in the same step, and does not continue u's recursion at its children. (c) In the final step, u chooses itself as a splitting node and informs its child. Afterwards, all solid nodes have been reached by u, and all dashed nodes have continued a more promising recursion instead.

Let T_σ be the subtree of T rooted at σ. The root u of T will take care of informing every node $v \in V[T] \setminus V[T_\sigma]$ about d_{uv} in the next recursion, whereas the task of informing the nodes $v \in V[T_\sigma]$ about d_{uv} is delegated to the children of σ. For that purpose, u informs σ about the distance $d_{u\,\sigma} = \hat{d}(u) + d_{G(u,i)}(u,\sigma)$. Subsequently, σ instructs each of its children c in T to start another recursion in their respective subtree T_c by sending it the distance $d_{uc} = d_{u\,\sigma} + w(\{\sigma\}, c)$ via the local edge $\{\sigma, c\}$. Note that we have to continue the recursion in the subtrees of the *children* of σ rather than the subtree of σ itself, because otherwise we are not able to guarantee that the trees' sizes halve in each step; this would only be true if the graph had constant degree.

Resolve Collisions Since *all* nodes try to propagate distance information within their shortest-path trees in parallel, σ may have been chosen as a splitting node by multiple roots. Therefore, sending $d_{u\,\sigma}$ to σ over the global network might require σ to receive more than $\mathcal{O}(\log n)$ messages. We carefully resolve this by making every root w of some tree T that intends to send a value $d_{w\,\sigma}$ to σ participate in an aggregation towards σ using the Aggregation Algorithm described in Chapter 4. Thereby, σ learns the minimum of all values $d_{w\,\sigma}$ sent to it, breaking ties by preferring the message from the node w that minimizes $\mathrm{sph}_{T(w,i)}(w,\sigma)$, or, if there are still ties, minimizes $\mathrm{id}(w)$. Note that in the aggregation process, the message for σ sent by the root u might be blocked by some other message, which disrupts the recursion intended by u to be continued at σ (this happens at node x in Figure 6.1a). However, as we will show shortly, continuing only the most promising recursion is sufficient for x to obtain the distance label minimizing Equation 6.1.

In the subsequent recursion level, every node v has to continue only the most promising recursion it has received so far, i.e., the recursion of the node w such that d_{wv} is minimized, breaking ties using $\mathrm{sph}_{T(w,i)}(w,v)$ and $\mathrm{id}(w)$ as described above (see Figures 6.1b and 6.1c). Since the subtree of each recursion halves in each recursion step, the process ends after $\mathcal{O}(\log n)$ steps. After the recursions have finished, every node v with $\mathrm{sph}(s,v) \leq t(i)$ has $\hat{d}(v) = d(s,v)$, which maintains Invariant (2) for the next iteration $i+1$. We state our main theorem, which we formally prove in the next section.

Theorem 6.2 (Exact SSSP)**.** *There is an algorithm that solves exact SSSP for any graph G in time $\widetilde{\mathcal{O}}(\sqrt{\mathsf{SPD}})$, w.h.p.*

We remark that the local capacity required by the algorithm is $\lambda = O(n^2)$, since in the worst case the *entire* graph needs to be sent over a node's incident edges.

6.1.2. Detailed Description

The focus of this section lies in a precise description of the propagation process outlined in Section 6.1.1 and an analysis of the overall algorithm. Assuming that the invariants in Proposition 6.1 hold at the beginning of some phase $i \in \{1, \dots, \lceil\sqrt{2\,\mathsf{SPD}}\rceil\}$, we describe the execution of the phase and prove that the invariants are maintained for the next phase, inductively establishing the correctness of our algorithm. Clearly, Invariant (1), which states that each node knows the subgraph of its $2(i-1)$-hop distance, and Invariant (2), which states that each node v with $\mathrm{sph}(s,v) \leq t(i-1)$ knows its correct distance to s, hold at the beginning of the first phase.

The pseudocode of our algorithm for phase i from the perspective of a node $v \in V$ can be found in Algorithm 1. The algorithm is divided into five steps, where the last three steps are repeated $\mathcal{O}(\log n)$ times. Step 1 ensures Invariant (1) by letting v introduce all edges of $G(v, 2i-2)$ to its neighbors. After performing the introduction twice, every node knows $G(v, 2i)$, and, specifically, $T(u,i)$ for every node $u \in V(v,i)$. Steps 2 to 5 make sure that Invariant (2) holds at the beginning of the next phase by employing a recursive divide-and-conquer approach. In the following, we describe how exactly the recursions are performed from the perspective of v.

Recursion Messages To begin the recursive approach, v first creates a *recursion message* $\langle v, \hat{d}(v), \emptyset\rangle$ in Step 2, which initiates a recursion on $T(v,i)$. Throughout the algorithm, v may take over the recursion of some other node by a recursion message that is sent to v. More precisely, a recursion message $\langle u, d_{uv}, L\rangle$ corresponds to a recursion that v currently handles, and is associated with a node u within hop-distance i to v that originally initiated the recursion on $T(u,i)$ (u might be v itself). The reason for v now storing this message is because in some preceding step, u instructed v to continue the recursion on a subtree T of $T(u,i)$ that is rooted at

Algorithm 1 `Exact-SSSP` ▷ Phase i executed by a node v

```
    Step 1
 1: send $G(v, 2i-2)$ to all neighbors, learn $G(v, 2i-1)$
 2: send $G(v, 2i-1)$ to all neighbors, learn $G(v, 2i)$

    Step 2
 3: $R \leftarrow \{\langle v, \hat{d}(v), \emptyset \rangle\}$        ▷ initial recursion message

 4: for $\lceil \log n \rceil + 1$ steps do        ▷ divide-and-conquer on subtrees
 5:     let $R = \langle u, d_{uv}, L \rangle$
 6:     $T \leftarrow$ subtree of $T(u, i)$ rooted at $v$ without the subtrees of the nodes of $L$
        Step 3
 7:     if $|V[T]| > 1$ then
 8:         $\sigma \leftarrow$ splitting node of $T$
 9:         $R_{cur} \leftarrow \langle u, d_{uv}, L \cup \{\sigma\} \rangle$        ▷ continue current recursion at $v$
10:         send splitting message $\langle u, d_{uv} + d_{T(u,i)}(v, \sigma), \mathrm{sph}_{T(u,i)}(u, \sigma) \rangle$ to $\sigma$
        Step 4
11:     if received splitting message $\langle u, d_{uv}, \mathrm{sph}_{T(u,i)}(u, v) \rangle$ then
12:         $\hat{d}(v) \leftarrow \min\{\hat{d}(v), d_{uv}\}$
13:         send recursion message $\langle u, d_{uv} + w(\{v, c\}), \emptyset \rangle$ to each child $c$ of $v$ in $T(u, i)$

        Step 5
14:     let $R_{new} = \langle u, d_{uv}, \emptyset \rangle$ be received recursion message with smallest
        distance value, breaking ties using smallest $\mathrm{sph}_{T(u,i)}(u, v)$ and $\mathrm{id}(u)$
15:     $\hat{d}(v) \leftarrow \min\{\hat{d}(v), d_{uv}\}$
16:     $R$ becomes $R_{cur}$ or $R_{new}$, whichever corresponds to the node $u$ that
        minimizes $d_{uv}$, breaking ties using smallest $\mathrm{sph}_{T(u,i)}(u, v)$ and $\mathrm{id}(u)$
```

v (we call T the corresponding *recursion subtree*). Further, the recursion message contains a value $d_{uv} = \hat{d}(u) + d_{T(u,i)}(u, v)$, which always corresponds to the length of a path from s to v that contains u. Finally, L is a subset of nodes of $T(u, i)$ from which v can infer T. More precisely, T is the subtree of $T(u, i)$ rooted at v without all subtrees rooted at any node of L. Note that whereas we cannot efficiently send T over the global network, T can be inferred from v's knowledge of $T(u, i)$ (which is fully contained in $G(v, 2i)$), and the set L, which will contain at most $\mathcal{O}(\log n)$ nodes. The initial recursion message $\langle v, \hat{d}(v), \emptyset \rangle$, therefore, instructs v to perform a recursion for v on the complete subtree $T(v, i)$.

Handling a Recursion Each recursion is handled through $\lceil \log n \rceil + 1$ *recursion steps*, which are performed by Steps 3 to 5 of our algorithm. Let $R = \langle u, d_{uv}, L \rangle$ be the recursion message corresponding to the recursion handled in some recursion step, and T be the subtree of $T(u, i)$ rooted at v without the subtrees rooted at the nodes of L. If $|V[T]| \leq 1$ (i.e., T contains at most one node), then there is no need to continue the recursion. Otherwise, the goal of Step 3 is to select a splitting node

σ whose removal disconnects T into components of size $|V[T]|/2$; these components become recursion subtrees in the next recursion step. Before we describe how the recursions can be continued in the resulting components, we show how the splitting node can be computed. Note that the computation is handled by v *locally* without necessitating any additional communication.

Lemma 6.3 (Splitting Node)**.** *Let T be a recursion subtree with $|V[T]| \geq 2$. Node v can compute a splitting node σ whose removal disconnects T into trees each of size at most $|V[T]|/2$.*

Proof. For an inner node u of T define $s(u)$ as the number of nodes in the subtree of T rooted at u, and let $p(u) = |V[T]| - s(u)$. The splitting node is computed by performing a search that descends into T, starting at its root v (which, as $|V[T]| \geq 2$, must have at least one child). If the search is currently at some inner node u, then let w be the child of u that maximizes $s(w)$ (choosing the node with minimum identifier in case of a tie). If $p(w) < |V[T]|/2$, then the search continues at w; otherwise, u is chosen as the splitting node. Note that if $p(w) < |V[T]|/2$, then w cannot be a leaf node. Clearly, the search can be performed locally at v and will eventually terminate at a splitting node σ.

Let y be the child of σ in S that maximizes $s(y)$. As σ is chosen as a splitting node, $p(y) \geq |V[T]|/2$. Therefore, $s(y) = |V[T]| - p(y) \leq |V[T]| - |V[T]|/2 = |V[T]|/2$, and, as y is the child that maximizes $s(y)$, the same holds for all children of σ.

If σ does not have a parent, then the claim holds immediately. Otherwise, its parent must have been considered as a splitting node as well. However, as it has not been chosen, $p(s) < |V[T]|/2$, which concludes the proof. ☐

Note that the previous lemma immediately implies that $\lceil \log n \rceil + 1$ recursion steps suffice until all recursion trees are of size 1. After having computed the splitting node σ, the recursion has to be continued in (1) $T \setminus T_\sigma$, which is rooted at v, and (2) all subtrees of T rooted at a child of σ in T (see the example in Figure 6.1a). To do so, v needs to send a recursion message to the root of each component.

To continue the recursion (1) in its own component $T \setminus T_\sigma$, v simply sends a recursion message $\langle u, d_{uv}, L \cup \{\sigma\}\rangle$ to itself. For (2), v needs to send a recursion message $\langle u, d_{uc}, \emptyset\rangle$, where $d_{uc} := d_{uv} + d_{T(u,i)}(v, c)$, to every child c of σ in T_σ. However, σ may have up to $\Theta(n)$ children; therefore, instead of sending recursion messages directly, v instructs σ to forward the respective messages to its children by sending a *splitting message* $\langle u, d_{u\sigma}, \mathrm{sph}_{T(u,i)}(u, \sigma)\rangle$ to σ, where

$$d_{u\sigma} = \hat{d}(u) + d_{T(u,i)}(u, \sigma) = d_{uv} + d_{T(u,i)}(v, \sigma)$$

is the length of a path from s to σ that contains u and v. Since σ can infer $T(u, i)$ after Step 1 of phase i, σ can reconstruct all recursion messages it is supposed to forward to its children in T_σ and send them directly using *local* edges.

Aggregate Splitting Messages As already pointed out in Section 6.1.1, v cannot send its splitting message to σ directly, as σ may be the recipient of many splitting messages in this recursion step. However, it suffices for every node to only receive the splitting message that contains the smallest distance value among all splitting messages destined at it. If there are several splitting messages with the same distance value, we prefer the one with smaller shortest path hop-distance value, breaking ties by choosing the message that contains the node with smallest identifier. In Lemma 6.4, we will argue that for every node v with $\text{sph}(s, v) \leq t(i)$ there must be a node u such that $\hat{d}(u) + d_{T(u,i)}(u, v) = d(s, v)$ and every recursion message corresponding to u that is destined at a node of the branch from u to v in $T(u, i)$ is successfully delivered; therefore, v will learn $d(s, v)$ anyway. Thus, the nodes send all splitting messages of this step using the Aggregation Algorithm of Theorem 4.3. More specifically, v is contained in the aggregation group $A_{\text{id}(\sigma)}$, and the aggregate function takes the minimum of all distance values, breaking ties as described above. Since each node is member of only one aggregation group, and target of at most one aggregation, the Aggregation Algorithm terminates within $\mathcal{O}(\log n)$ rounds, w.h.p.,

Deliver Recursion Messages Thereby, v may receive a splitting message $\langle u, d_{uv}, \text{sph}_{T(u,i)}(u, v)\rangle$ that corresponds to the recursion of some node u (but not necessarily *from* u). In this case, in Step 4 v updates $\hat{d}(v)$ to $\min\{\hat{d}(v), d_{uv}\}$ and sends the corresponding recursion messages to its children in $T(u, i)$, using its knowledge of $T(u, i)$ due to Step 1. If v is adjacent to multiple nodes that have received splitting messages, then in Step 5 it may receive multiple recursion messages. However, it again suffices for v to only store the recursion message that contains the smallest distance value. We again break ties by preferring the recursion message associated with the node u that minimizes $\text{sph}_{T(u,i)}(u, v)$, breaking ties using the node's identifiers. Finally, v updates $\hat{d}(v)$ considering the received message's distance. Furthermore, it decides whether to continue its current recursion, or the recursion that corresponds to a received recursion message, whichever minimizes d_{uv} (breaking ties as before).

The next lemma shows that our algorithm maintains Invariant (2).

Lemma 6.4 (Invariant (2))**.** *Let $v \in V$ such that sph$(s, v) \leq t(i - 1)$. At the beginning of phase i we have that $\hat{d}(v) = d(s, v)$.*

Proof. We prove by induction on i. The statement obviously holds at the beginning of the first phase as $t(0) = 0$.

Now consider phase $i > 1$ and let $v \in V$ such that $\text{sph}(s, v) \leq t(i)$. There must exist a node u that lies on some shortest path from s to v such that $\text{sph}(s, u) \leq t(i-1)$ and $\text{sph}(u, v) \leq i$. By the induction hypothesis, $\hat{d}(u) = d(s, u)$. If there are multiple nodes u that lie on a shortest path from s to v and for which $\hat{d}(u) = d(s, u)$ hold, we choose u to be the one that minimizes $\text{sph}(u, v)$; if there are also multiple of those, choose the one with smallest identifier. Note that v lies in $T(u, i)$; further, $\text{sph}_{T(u,i)}(u, v) = \text{sph}(u, v) \leq i$. We argue that every recursion message or splitting message u intends to send to any node x on the branch P from u to v in $T(u, i)$ is successfully delivered. Since the branch from u to v is a shortest path from u to v that contains at most i hops, the value of the recursion message u wants to send to x is

$$d_{ux} = \hat{d}(u) + d_{T(u,i)}(u, x) = d(s, u) + d(u, x) = d(s, x).$$

Therefore, x will never receive any message with an even smaller distance value, since this would imply that there is a shorter path from s to x.

Thus, assume that some node of P *sends* a message for u containing the distance value d_{ux}, but x does not *receive* the message because a message corresponding to the recursion of a node u' with the same distance value $d_{u'x} = d_{ux}$ is preferred. By definition of the algorithm, the message corresponding to the recursion of u' must have been preferred because (1) $\text{sph}_{T(u',i)}(u', x) < \text{sph}_{T(u,i)}(u, x)$, or (2) $\text{sph}_{T(u',i)}(u', x) = \text{sph}_{T(u,i)}(u, x)$, but $\text{id}(u') < \text{id}(u)$. In both cases, u' lies on a shortest path from s to x; further, $\hat{d}(u') = d(s, u')$, since $d_{u'x} = d_{ux}$ and the shortest path from s to x containing u' would become even shorter if $\hat{d}(u')$ would still be too large in this phase. In Case (1), there is a shortest path from u' to v over x that contains fewer hops than the path from u to v in $T(u, v)$, which contains $\text{sph}(u, v) \leq i$ hops by definition of $T(u, i)$. In Case (2), there exist shortest paths from both u and u' to x that contain the same number of hops, but u' has a smaller identifier. Thus, in both cases, we must haven chosen u' over u, which contradicts our choice of u.

Analogously, we can argue that any node x on P that ceases to continue a recursion for u does so because it prefers to continue a recursion for some node u'. In this case, u must either yield a shorter distance from s to x, or must be within smaller hop-distance to x, leading to a contradiction as above. □

Termination As we do not require the nodes to know SPD, we have to let the nodes detect when to terminate. We simply stop the algorithm when for the first time no distance estimate changes at any node, which can be detected by performing the Aggregate-and-Broadcast Algorithm at the end of every phase. When for the first time no value changes anymore, all nodes terminate.

Lemma 6.5 (Termination)**.** *No distance estimate changes in phase i if and only if $\hat{d}(v) = d(s, v)$ for every node $v \in V$.*

Proof. First, if $\hat{d}(v) = d(s, v)$ for every node v, then clearly v will never receive a smaller distance value in any recursion message anymore, as in that case there would exist an even shorter path from s to v.

For the other direction, assume that no distance estimate changes in some phase i. Let $v \in V$. We prove that $\hat{d}(v) = d(s, v)$ already at the beginning of phase i by induction on $\text{sph}(s, v)$. If $\text{sph}(s, v) = 0$, then $v = s$, and as all edge weights are positive, $\hat{d}(v) = d(s, v)$ already at the beginning of the first phase, and thus also at the beginning of phase i. Now let $\text{sph}(s, v) \geq 1$. Then there exists a neighbor u of v such that $\text{sph}(s, u) = \text{sph}(s, v) - 1$ and $d(s, v) = d(s, u) + w(\{u, v\})$. By the induction hypothesis, $\hat{d}(u) = d(s, u)$ at the beginning of phase i, and by definition of our algorithm, v must receive $d(s, v)$ as a distance value in a recursion message throughout the execution of phase i. However, as $\hat{d}(v)$ does not change in phase i, $\hat{d}(v) = d(s, v)$ already at the beginning of phase i. □

Lemma 6.6 (Runtime)**.** *The algorithm terminates after $\lceil\sqrt{2\,\mathsf{SPD}}\rceil$ phases. Every phase takes time $\mathcal{O}(\log^2 n)$, w.h.p.*

Proof. As any shortest path has length at most SPD, after phase $\lceil\sqrt{2\,\mathsf{SPD}}\rceil$ every node u knows $d(s, u)$ by Lemma 6.4. Therefore, Lemma 6.5 implies that no distance value changes in the subsequent round, in which case the algorithm terminates. In each of the $\lceil\log n\rceil + 1$ recursion steps, each node is member and target of at most one aggregation group, and the aggregation takes time $\mathcal{O}(\log n)$ by Theorem 4.3, w.h.p. By Theorem 4.1, the Aggregate-and-Broadcast Algorithm to detect termination at the end of a phase takes an additional $\mathcal{O}(\log n)$ rounds. □

The lemmas above immediately imply Theorem 6.2. Finally, we remark that the algorithm can easily be modified to solve (h, k)-SSP for given h and k.

Theorem 6.7 ((h, k)-SSP)**.** *The modified algorithm solves (h, k)-SSP in time $\widetilde{\mathcal{O}}(\sqrt{kh})$, w.h.p.*

Proof. As the shortest-path diameter of G is generally not known, our SSSP algorithm has to incrementally increase the distance at which the nodes learns their respective neighborhood and propagate their distance information. We modify this as follows. For a given hop-distance h, we can first let every node $v \in V$ learn $G(v, 2\lceil\sqrt{kh}\rceil)$ by sending all information about G known so far via its local edges for $2\lceil\sqrt{kh}\rceil$ rounds. Then, we separately perform $\lceil h/\sqrt{kh}\rceil$ phases of the algorithm for each k, where in each phase every node v always propagates distance information by $\lceil\sqrt{kh}\rceil$ hops. The total runtime amounts to $\widetilde{\mathcal{O}}(\sqrt{kh} + kh/\sqrt{kh}) = \widetilde{\mathcal{O}}(\sqrt{kh})$, which concludes the theorem. □

6.2. Algorithm for Approximate SSSP

Finally, we present a $(\log_\alpha n)^{\mathcal{O}(\log_\alpha n)}$-approximate SSSP algorithm that takes time $\widetilde{\mathcal{O}}(\alpha^3)$, w.h.p., for a parameter $\alpha \geq 5$. By setting $\alpha = n^{\varepsilon/3}$ for some $\varepsilon > 0$, we obtain a $(1/\varepsilon)^{\mathcal{O}(1/\varepsilon)}$-approximation in time $\widetilde{\mathcal{O}}(n^\varepsilon)$. We first describe the algorithm from a high level and provide all details in Section 6.2.2.

6.2.1. Overview

The main idea of the algorithm is to recursively construct a hierarchy of graphs $G_1, \dots, G_T$, where $V[G_i] \subseteq V[G_{i-1}]$. The set $M_i := V[G_i]$ contains a node of G_{i-1} with probability $\log(n)/\alpha$ for $i = 2$, and with probability $1/\alpha$ for $i \geq 3$. The first spanner G_1, which contains all nodes of G, is constructed using only the local network by simply performing the distributed spanner algorithm of Baswana and Sen [BS07] with parameter k as a black box, which gives a $(2k-1)$-spanner in time $\mathcal{O}(k^2)$.

The construction of the subsequent spanners $G_2, \dots, G_T$ relies entirely on the global network. For $i \geq 2$, we construct G_i as an *h-hop skeleton spanner* of G_{i-1}. Roughly speaking, the skeleton spanner is a spanner of a *skeleton* of G_{i-1} rather than a spanner of G_{i-1} itself, where the skeleton of G_{i-1} is a graph that contains only some of the nodes of G_{i-1}, and an edge between every two nodes that are sufficiently close in G_{i-1}. Therefore, a skeleton spanner contains edges that are not necessarily edges of G_{i-1}. However, we construct these edges such that for every two nodes in G_{i-1} that are within hop-distance h, the skeleton spanner contains a path that gives a good distance approximation. Furthermore, the skeleton spanner has low arboricity. This allows us to let every edge of the spanner G_i be learned by only *one* endpoint of the edge while ensuring that no node has to take care of more than $\widetilde{\mathcal{O}}(\alpha)$ edges. On this graph, we can efficiently apply the techniques of

Chapter 4 using the global network. More specifically, we will prove that we can construct G_i as an $\mathcal{O}(\alpha)$-hop skeleton spanner of stretch $\mathcal{O}(\log_\alpha n)$ of the graph G_{i-1} in time $\widetilde{\mathcal{O}}(\alpha^3)$ for all $i \geq 2$.

Finally, by taking the union of all the graphs G_i, we obtain a graph that approximates all distances well. More precisely, by applying the properties of the skeleton spanners G_i, we show that H has a $(\log_\alpha n)^{\mathcal{O}(\log_\alpha n)}$-approximate path consisting of at most $\widetilde{\mathcal{O}}(\alpha)$ hops for every pair of nodes $u, v \in V$. Therefore, every node v learns a good approximation $\tilde{d}(s, v)$ of $d(s, v)$ by performing a distributed Bellman-Ford algorithm with source s in H for $\widetilde{\mathcal{O}}(\alpha)$ iterations. The low arboricity of H allows us to perform each iteration of Bellman-Ford in the global network in time $\widetilde{\mathcal{O}}(\alpha)$. The following theorem results from a careful analysis of the approximation guarantees and runtime of our recursive spanner construction, and is formally proven in the following section.

Theorem 6.8 (Approximate SSSP). *There is an algorithm that solves $(\log_\alpha n)^{\mathcal{O}(\log_\alpha n)}$-approximate SSSP in time $\widetilde{\mathcal{O}}(\alpha^3)$, w.h.p.*

6.2.2. Detailed Description

In the first part of this section, we present an algorithm to compute a skeleton spanner, which is the key ingredient for our approximation algorithm. Subsequently, we describe how this algorithm can be used to compute $(\log_\alpha n)^{\mathcal{O}(\log_\alpha n)}$-approximate SSSP by constructing a hierarchy of skeleton spanners.

Constructing a Skeleton Spanner We first define skeleton spanners formally in Definition 6.9 and subsequently describe the algorithm to compute such a spanner from a high level. As the algorithm is solely based on computing *limited-depth Bellman-Ford computations*, we can efficiently execute it in the global network by using our methods from Chapter 4.

Definition 6.9 (Skeleton Spanner). *Let $G = (V, E, w)$ be a weighted graph, let $M \subseteq V$ be a set of marked nodes of G, and let $h \in \mathbb{N}$. An h-*hop skeleton spanner* $H = (M, E_H)$ with stretch $s \geq 1$ is a weighted graph with the following properties:*

1. *Every edge $\{u, v\} \in E_H$ corresponds to a path P in G between u and v such that $w(\{u, v\}) = w(P)$.*
2. *For every two nodes $u, v \in M$, we have $d_G(u, v) \leq d_H(u, v) \leq s \cdot d_{h,G}(u, v)$.*

Note that an h-hop skeleton spanner approximates nodes within hop-distance h well, whereas it does not give any guarantees for nodes that are farther away from each other.

From a high level, our algorithm to construct an h-hop skeleton spanner works as follows. Assume that we are given a graph $G = (V, E, w)$, a set of marked nodes $M \subseteq V$, and a hop-distance parameter $h \geq 1$. Let us further assume that for all $e \in E$, we have $1 \leq w(e) \leq W/h$ for some given $W \geq h$, so that the length of any path consisting of at most h hops is between 1 and W. The algorithm further has two parameters $k \geq 2$ and $\eta > 1$ that control the stretch and the number of edges of the resulting spanner.

Algorithm 2 `Skeleton-Spanner` ▷ $\eta > 1$, $k \geq 2$
▷ *Stage i dealing with h-lim. distances $\in [L_i/\eta, L_i]$*

```
1: $V_0 := V$                                              ▷ $V_j$ is set of nodes active in phase $j$
2: for $j := 0$ to $k-1$ do
3:     $G_j := G[V_j]$                                    ▷ subgraph of $G$ induced by $V_j$
4:     for each $r \in V_j \cap M$ do                     ▷ $M$ is set of marked nodes of $G$
5:         randomly sample $r$ with probability $|M|^{\frac{j+1}{k}-1}$ into set $R_j$
6:     for each $r \in R_j$ do
7:         for each $v \in B_{G_j}(r, k-j, L_i) \cap M$ do
8:             add $\{v, r\}$ of weight $d_{h(k-j),G_j}(v,r)$ to $E_H$
9:     $V_{j+1} := V_j \setminus \left(\bigcup_{r \in R_j} B_{G_j}(r, k-j-1, L_i)\right)$
```

The algorithm consists of $\lceil \log_\eta W \rceil$ stages. In the following, we focus on a specific stage $i \geq 1$. For convenience, we define $L_i := \eta^i$. The objective of stage i is to construct a subset of the edges of $H = (M, E_H)$ that provides a good approximation for any two nodes $u, v \in M$ for which the h-limited distance in G is in the range $[L_i/\eta, L_i]$. The final spanner is then obtained by taking the union of the edges computed in the individual stages.

Each stage consists of k phases, which we enumerate by $j = 0, 1, \ldots, k-1$. Initially, all nodes in M are active. We will show that nodes in M become inactive as soon as it is guaranteed that all their h-limited distances in the target range are approximated sufficiently well. In the following, for a node $r \in G$, an integer parameter $x \geq 1$, and a distance $L \geq 1$, we define the *ball* of r as

$$B_G(r, x, L) := \{v \in V[G] \mid d_{h \cdot x, G}(r, v) \leq x \cdot L\}.$$

The details of the algorithm for stage i are given in Algorithm 2. We refer to the k iterations of the outermost for-loop as the k phases $j = 0, \ldots, k-1$. Furthermore, we say a node $v \in V_j$ gets *deactivated* in phase j of stage i if it is contained in the ball $B_{G_j}(r, k-j-1, L_i)$ of some sampled node $r \in R_j$. Note that when a node gets deactivated, it will not participate in any subsequent phase of stage i.

The following sequence of lemmas proves that the algorithm constructs an h-hop skeleton spanner.

Lemma 6.10 (Skeleton Spanner Stretch)**.** *When a node $u \in M$ gets deactivated in stage i, for every $v \in M$ for which $d_{h,G}(u,v) \leq L_i$, the algorithm has added a path of length at most $2kL_i$ to the spanner edge set E_H. Furthermore, this path consists of at most 2 edges.*

Proof. Let $u, v \in M$ be two nodes for which $d_{h,G}(u,v) \leq L_i$ and let us show that the algorithm adds a path between u and v of length at most $2kL_i$ and consisting of at most 2 edges to the spanner; in the following, we call such a path a (≤ 2)-hop path. W.l.o.g., assume that u is deactivated in phase j and that v is deactivated in a phase $j' \geq j$. If the algorithm has already added a (≤ 2)-hop path of length at most $2kL_i$ between u and v prior to phase j, we are done. Otherwise, we show that (1) $d_{h,G_j} \leq L_i$ in the graph G_j of the active nodes in phase j, and (2) the algorithm adds a (≤ 2)-hop path of length at most $2kL_i$ between u and v to the skeleton spanner in phase j.

Let P be a path between u and v in G that contains $|P| \leq h$ hops and has length $w(P) \leq L_i$. First assume that all nodes of P are still active in phase j. We then clearly have $d_{h,G_j}(u,v) \leq L_i$. As node u gets deactivated in phase j, we have that $u \in B_{G_j}(r, k-j-1, L_i)$ for some sampled node $r \in R_j$. We thus add an edge $\{u, r\}$ of weight $d_{h(k-j-1),G_j}(r,u) \leq (k-j-1)L_i$ to E_H. Because $u \in B_{G_j}(r, k-j-1, L_i)$ and $d_{h,G_j}(u,v) \leq L_i$, we can further conclude that $v \in B_{G_j}(r, k-j, L_i)$. We thus also add an edge (v, r) of weight $d_{h(k-j),G_j}(r,v) \leq (k-j)L_i$ to E_H. Together, the two edges provide a (≤ 2)-hop path of length at most $(2(k-j)-1)L_i < 2kL_i$ between u and v.

It remains to consider the case that some nodes of P are deactivated before phase j. Let $j' < j$ be the first phase in which some node of P is deactivated, and let $w \in V$ be some node of P that is deactivated in phase j'. This implies that there is some sampled node $r' \in R_{j'}$ such that $w \in B_{G_{j'}}(r, k-j'-1, L_i)$. Because the path P is completely contained in $G_{j'}$, we have $d_{h,G_{j'}}(w,u) \leq L_i$ and $d_{h,G_{j'}}(w,v) \leq L_i$. Therefore, both nodes u and v are contained in $B_{G_{j'}}(r, k-j', L_i)$, and thus in phase j', the algorithm adds edges $\{u, r'\}$ and $\{v, r'\}$ of weight at most $(k-j')L_i \leq kL_i$ to E_H. These edges form a (≤ 2)-hop path of length at most $2kL_i$ between u and v in the constructed spanner. □

The next lemma shows that in each phase, every node is only within the ball of a few random centers. On the one hand, this implies that the spanner algorithm does not add too many edges; on the other hand, it also allows to execute the algorithm efficiently by using only global edges. The lemma follows because the radius of the balls decreases from phase to phase, and the radius in which nodes are deactivated in phase j is the same as the radius of the ball of nodes to which edges are established in phase $j+1$. Therefore, roughly speaking, if a node v lies within the balls of many sampled centers in phase $j+1$ (to which it would establish an edge), the node should have already been deactivated in phase j. A similar argument has previously been used by Blelloch et al. [Ble+14].

Lemma 6.11 (Ball Congestion). *For phase j of Algorithm 2 and $v \in V_j$, define*

$$\mathcal{R}_{v,j} := \{r \in R_j \mid v \in B_{G_j}(r, k-j, L_i)\}.$$

We have that $|\mathcal{R}_{v,j}| = \mathcal{O}(|M|^{1/k} \log n)$, w.h.p.

Proof. Consider phase $j \geq 0$ and let $v \in V_j$. We define $R_{v,j} = \{r \in M \mid v \in B_{G_j}(r, k-j, L_i)\}$ and $R'_{v,j} = \{r \in M \mid v \in B_{G_j}(r, k-j-1, L_i)\}$. The nodes in $R_{v,j}$ are the ones to which, if sampled in phase j, v establishes an edge to, whereas $R'_{v,j}$ contains the set of nodes that, when sampled in phase j, deactivate v. In phase j, nodes of M are sampled with probability $p_j := |M|^{\frac{j+1}{k}-1}$. We need to show that, w.h.p., $p_j \cdot |R_{v,j}| = \mathcal{O}(|M|^{1/k} \cdot \log n)$ for all $v \in V$ and all phases j. Since this value is an upper bound on the expected size of $\mathcal{R}_{v,j}$, which can be modeled as a sum of independent binary random variables, we can then use a Chernoff bound to show that $|\mathcal{R}_{v,j}| = \mathcal{O}(|M|^{1/k} \log n)$, w.h.p. The lemma then follows by taking the union bound over all v and j.

To prove that $p_j \cdot |R_{v,j}| = \mathcal{O}(|M|^{1/k} \cdot \log n)$, we show that otherwise, v would have been deactivated in the previous phase, w.h.p. In the following, let $c > 0$ be a constant that will be determined at the end. For a node $v \in V_j$ and a phase j, let $\mathcal{E}_{v,j}$ be the event that $p_j \cdot |R'_{v,j}| \geq c \ln n$ and that node v is *not* deactivated in phase j. Recall that node v is deactivated in phase j if and only if one of the nodes in $R'_{v,j}$ is sampled in Algorithm 2. For all $v \in V_j$, we therefore have

$$\Pr(\mathcal{E}_{v,j}) \leq (1-p_j)^{|R'_{v,j}|} < e^{-p_j|R'_{v,j}|} \leq \frac{1}{n^c}. \tag{6.2}$$

Note that for $v \in V_j$ for which $p_j \cdot |R'_{v,j}| < c \ln n$, we have $\Pr(\mathcal{E}_{v,j}) = 0$. Let us further define the random variable $X_{v,j}$ as the number of sampled nodes $r \in R_j$ in phase j for which node $v \in V_j$ is in $B_{G_j}(r, k-j, L)$. That is, $X_{v,j}$ counts the number of sampled nodes from $R_{v,j}$ in phase j. We show that $X_{v,j} = \mathcal{O}(|M|^{1/k} \log n)$, w.h.p.

To study the number of sampled nodes from $R_{v,j}$, observe that $R_{v,j} \subseteq R'_{v,j-1}$, which follows from the definitions of $R_{v,j}$ and $R'_{v,j}$ and the fact that G_j is a subgraph of G_{j-1}. If we condition on the event $\overline{\mathcal{E}}_{v,j-1}$, we know that $p_{j-1}|R'_{v,j-1}| < c \ln n$, as otherwise v would have been deactivated in phase $j-1$ and thus $v \notin V_j$. The sampling probabilities increase by a factor $|M|^{1/k}$ from phase to phase and thus, conditioning on $\overline{\mathcal{E}}_{v,j-1}$ implies that $p_j|R_{v,j}| \leq p_j|R'_{v,j-1}| < c|M|^{1/k} \ln n$. We therefore have $\mathbb{E}[X_{v,j}] < c|M|^{1/k} \ln n$. The Chernoff bound of Lemma 2.2 implies that

$$\begin{aligned} &\Pr\left(X_{v,j} \geq 2c|M|^{1/k} \ln n \,\middle|\, \overline{\mathcal{E}}_{v,j-1}\right) \\ &\leq e^{-c/3 \cdot |M|^{1/k} \cdot \ln n} \overset{(|M|^{1/k} > 1)}{<} \frac{1}{n^{c/3}}. \end{aligned} \tag{6.3}$$

Let $\mathcal{B}_{v,j}$ be the event that $X_{v,j} \geq 2c|M|^{1/k} \ln n$. By using the law of total probability, we then get

$$\begin{aligned} \Pr(\mathcal{B}_{v,j}) &= \Pr\left(\mathcal{B}_{v,j} \,\middle|\, \overline{\mathcal{E}}_{v,j-1}\right) \cdot \Pr(\overline{\mathcal{E}}_{v,j-1}) \\ &\quad + \Pr\left(\mathcal{B}_{v,j} \,\middle|\, \mathcal{E}_{v,j-1}\right) \cdot \Pr(\mathcal{E}_{v,j-1}) \\ &\leq \Pr\left(\mathcal{B}_{v,j} \,\middle|\, \overline{\mathcal{E}}_{v,j-1}\right) + \Pr(\mathcal{E}_{v,j-1}) \\ &\overset{(6.2),(6.3)}{\leq} \frac{1}{n^{c/3}} + \frac{1}{n^c}, \end{aligned}$$

which concludes the proof. □

We now have everything we need in order to prove the main property of the described spanner algorithm.

Lemma 6.12 (Correctness of Algorithm 2)**.** *Given a weighted graph $G = (V, E, w)$, a set of marked nodes $M \subseteq V$, as well as parameters $h \geq 1$, $k \geq 2$, and $\eta > 1$, the described spanner algorithm computes an h-hop skeleton spanner $H = (M, E_H)$ with the following properties:*

1. *H has stretch $2\eta k$.*
2. *$|E_H| = O(k \cdot |M|^{1+1/k} \log n \cdot \log_\eta W)$.*
3. *For any two nodes $u, v \in M$ such that $hop_G(u, v) \le h$ we have that*
$$d_{2,H}(u, v) \le 2\eta k \cdot d_{h,G}(u, v),$$
i.e., there exists a path P with at most 2 hops and length at most $2\eta k \cdot d_{h,G}(u, v)$.

All properties hold w.h.p.

Proof. First note that by construction, we have $d_H(u, v) \ge d_G(u, v)$ for all $u, v \in M$ as the weight of every edge that we add to H corresponds to a path in G. Also note that at the end of Algorithm 2, all nodes are deactivated. This follows because for $j = k - 1$, the sampling probability is set to 1 and therefore in the last phase, each remaining node in M is a sampled.

To prove the stretch bound, it remains to show that $d_H(u, v) \le 2\eta k \cdot d_{h,G}(u, v)$. We will at the same time also show that any two nodes within hop-distance h in G will be connected by a (≤ 2)-hop path of this length. We first consider a single stage i. Lemma 6.10 together with the fact that at the end, all nodes are inactive, implies that for any two nodes $u, v \in M$ for which $d_{h,G}(u, v) \le L_i$, the spanner contains a (≤ 2)-hop path of length at most $2kL_i$. This provides a path of the right stretch for all node pairs $u, v \in M$ for which $d_{h,G}(u, v) \in [L_i/\eta, L_i]$. The stretch bound now directly follows because the spanner is defined as the union of the parts computed in each stage and because $\bigcup_{i=1}^{\lceil \log_\eta W \rceil} [L_i/\eta, L_i] \supseteq [1, W]$.

To upper bound the number of edges of the spanner H, we again consider a single stage i. In each phase j of stage i, each node $v \in V_j \cap M$ adds an edge to each node $r \in \mathcal{R}_{v,j} = \{r \in R_j \mid v \in B_{G_j}(r, k - j, L_i)\}$. By Lemma 6.11, the size of $\mathcal{R}_{v,j}$ is $\mathcal{O}(|M|^{1/k} \log n)$, w.h.p. As the stage has k phases, w.h.p., we therefore add at most $\mathcal{O}(k|M|^{1/k} \log n)$ edges per node $v \in M$ and thus at most $\mathcal{O}(k|M|^{1+1/k} \log n)$ edges in total. The lemma now follows because the total number of stages is $\mathcal{O}(\log_\eta W)$. □

Realization in the Global Network We will now show how the algorithm can be performed in the global network using our techniques from Chapter 4. To do so, we require that the graph G on which we aim to construct an h-hop skeleton spanner H is known by the nodes in a specific way. Formally, we define a graph G' to be in *δ-oriented form*, if every edge $\{u, v\} \in E[G']$ is only known by one of its endpoints (we say that endpoint is *responsible* for the edge), such that every node in $V[G']$ is responsible for at most δ edges. We also construct H in such a form; more specifically, when $v \in B_{G_j}(r, k - j, L_i) \cap M$ for some $r \in R_j$ in phase j and stage i of the algorithm, the edge $\{v, r\}$ is added to E_H by v and without the knowledge of r, and v becomes responsible for the edge.

To implement the algorithm efficiently in the global network, we perform multi-aggregations using a variant of Theorem 4.13. More precisely, instead of letting nodes in G_j communicate with their neighbors directly, we only let nodes communicate via multi-aggregations. To be able to do so, in phase j each node needs to have a *broadcast tree* $T_{v,j}$ that connects v with its neighbors in G_j. Using the notation of

Chapter 4, $T_{v,j}$ is the multicast tree of the multicast group $A_{\text{id}(v)} = N_{G_j}(v)$ with source $v \in V_j$. Recall that the *congestion* of the trees indicates the communication load every node has to perform when using the trees to perform multi-aggregations. The following lemma shows that, if a graph G_j in phase j of Algorithm 2 is in δ-oriented form, broadcast trees with low congestion can be efficiently constructed.

Lemma 6.13 (Realizing a Phase)**.** *Let G_j be the graph considered in phase j of Algorithm 2, and suppose that G_j is in δ-oriented form. There is an algorithm that constructs a multicast tree $T_{v,j}$ for every source $v \in V_j$ with multicast group $N_{G_j}(v)$ with congestion $\mathcal{O}(\delta + \log n)$ in time $\mathcal{O}(\delta + \log n)$, w.h.p. The algorithm ensures that each leaf of $T_{v,j}$ that corresponds to v's neighbor u learns $w(\{u, v\})$. Additionally, the algorithm brings G_{j+1} into δ-oriented form.*

Proof. We follow the idea of Lemma 4.29 and let every node $v \in V_j$ only contribute a packet $p_{v,\text{id}(u)}$ for the aggregation towards $u \in N_{G_j}(v)$, if v is responsible for $\{v, u\} \in E[G_j]$. By additionally inserting a packet $p_{u,\text{id}(v)}$ for the aggregation towards itself, all broadcast trees are properly constructed. Since each node only contributes δ packets, the construction process takes time $\mathcal{O}(\delta+\log n)$, w.h.p. As the total number of packets is bounded by δn, the resulting congestion is $\mathcal{O}(\delta + \log n)$, w.h.p (see Theorem 4.11). By annotating each packet with the weight of the respective edge, we ensure that the corresponding leaf learns the edge.

It remains to make sure that G_{j+1} is in δ-oriented form. That is, when a node u becomes deactivated in phase j, the endpoint that is responsible for every edge incident to u needs to exclude the edge from the graph. For all edges u is responsible for, u needs to do nothing; for all the others, it needs to inform the respective endpoints so that they can exclude the edge. To do so efficiently, we construct a multicast tree that connects u with all neighbors that are responsible for an edge incident to u, and use a multicast to inform them about whether u became deactivated. Since every node needs to join at most δ multicast groups, Theorems 4.11 and 4.12 imply that the multicast can be performed in time $\mathcal{O}(\delta + \log n)$, w.h.p. $\square$

We now describe exactly how multi-aggregations can be used to execute Algorithm 2. Specifically, we need to ensure that in phase j of stage i every node $v \in V_j \cap M$ learns $d_{h(k-j),G_j}(v, r)$ for each sampled $r \in R_j$ if $d_{h(k-j),G_j}(v, r) \le (k - j)L_i$. From a high level, we perform a distributed Bellman-Ford algorithm in G_j from every source $r \in M$ that is sampled in phase j. More precisely, in iteration t of the algorithm, every node $v \in V_j$ learns $d_{t,G_j}(r, v)$ for every node $r \in R_{v,t}$, where

$$R_{v,t} := \{u \in R_j \mid d_{t,G_j}(r, v) \le (k - j)L_i\}.$$

Therefore, after $h(k - j)$ iterations, v knows whether it is in $B_{G_j}(r, k - j, L_i)$ for all $r \in R_j$, in which case it adds $\{v, r\}$ to E_H and becomes responsible for the edge. Further, v learns whether there exists a node $r \in R_j$ such that $v \in B_{G_j}(r, k-j-1, L_i)$, in which case v becomes deactivated.

In iteration t of the algorithm, every node v performs a (slightly modified) multi-aggregation to inform each of its neighbors about $d_{t-1,G_j}(r, v)$ for every node $r \in R_{v,t}$. Our algorithm is essentially an extension of the distributed Bellman-Ford approach described in Section 4.4.2, where every node sends its current distance value to its neighbors using a multi-aggregation, and the leaf nodes augment the sent distance values by the corresponding edge's weights before aggregating the minimum value at each node. More precisely, v first sends the value $d_{t-1,G_j}(r,v)$ to all leaf nodes in its broadcast tree $T_{v,j}$. Using its knowledge of $w(\{v,u\})$ due to Lemma 6.13, every leaf node $l_{u,\text{id}(v)}$ of $T_{v,j}$ computes $d_{t-1,G_j}(r,v) + w(\{v,u\})$ before aggregating the minimum value that is sent towards u for r. Thereby, v receives

$$\min_{u \in V_j} \left(d_{t-1,G_j}(r,u) + w(u,v)\right) = d_{t,G_j}(r,v)$$

for each $r \in R_{v,t}$. Unlike the approach of Section 4.4.2, where distances to a *single* source are computed, here we have to make sure that the distances for *all* $r \in R_{v,t}$ reach v. To do so, we slow down the multi-aggregation by a factor linear in $|R_{v,t}|$, where we exploit the fact that $|R_{v,t}| \leq |\mathcal{R}_{v,j}| = \mathcal{O}(|M|^{1/k} \log n)$, w.h.p., by Lemma 6.11. Furthermore, distances that grow larger than $(k-j)L_i$ will simply be dropped by the respective leaf node.

More precisely, our modification of the Multi-Aggregation Algorithm described in Section 4.1.4 works as follows. Instead of sending a single packet of size $\mathcal{O}(\log n)$ to the root $r_{\text{id}(v)}$ of $T_{v,j}$, which is then multicast to the leaf nodes of $T_{v,j}$ in the butterfly, v sends one *subpacket* for each $r \in R_{v,t-1}$ containing the value $d_{t-1,G_j}(r,v)$. Subpackets are forwarded sequentially throughout the multi-aggregation, such that each round of forwarding packets in the butterfly is "simulated" by performing multiple rounds of forwarding subpackets. As the number of subpackets a packets consists of may vary, we synchronize each simulated round by performing the Aggregate-and-Broadcast algorithm, which introduces an additional $\mathcal{O}(\log n)$ time overhead per simulated round. When a packet from v reaches a leaf $l_{u,\text{id}(v)}$ of $T_{v,j}$ that corresponds to a node $u \in N_{G_j}(v)$, $l_{u,\text{id}(v)}$ removes all subpackets whose value, increased by $w(\{u,v\})$, exceeds $(k-j)L_i$. The subpackets are then mapped to random nodes of the butterfly, and subsequently aggregated towards their targets as in the original Multi-Aggregation Algorithm.

Lemma 6.14 (Realizing Algorithm 2)**.** *Let $G = (V,E)$ be a weighted graph given in δ-oriented form, and $M \subseteq V$. There is an algorithm that constructs an h-hop spanner with the properties described in Lemma 6.12 in $\mathcal{O}(k \cdot |M|^{1/k} \log n \cdot \log_\eta W)$-oriented form. The algorithm takes time*

$$\mathcal{O}((\delta + \log n)|M|^{1/k} \log n \cdot hk^2 \cdot \log_\eta W), \textit{ w.h.p.}$$

Proof. The correctness of the algorithm follows from the observation that in iteration t of the Bellman-Ford algorithm, every node $v \in V_j$ learns $d_{t,G_j}(r,v)$ for every node $r \in R_{v,t}$. From the proof of Lemma 6.12 it directly follows that every node adds at most $\mathcal{O}(k \cdot |M|^{1/k} \log n \cdot \log_\eta W)$ edges throughout the entire algorithm's execution, w.h.p.

It remains to show the runtime of the algorithm. Consider phase j of stage i. By Lemma 6.13, we can ensure that all broadcast trees $T_{v,j}$ are constructed at the beginning of phase j in time $\mathcal{O}(\delta + \log n)$, w.h.p. If each packet was of size $\mathcal{O}(\log n)$, then a single iteration of the Bellman-Ford computation would require time $\mathcal{O}(\delta + \log n)$ by Corollary 4.30. However, this is slowed down by the maximum number of subpackets a packet may consist of, which, as argued before, is bounded by $|R_{v,t}| = O(|M|^{1/k} \log n)$ due to Lemma 6.11, w.h.p. This runtime also covers the additional $\mathcal{O}(\log n)$ factor to synchronize each simulated round. Therefore, a single iteration takes time $\mathcal{O}((\delta + \log n)|M|^{1/k} \log n)$, w.h.p. In each phase, up to hk iterations of message passing are performed. Multiplying this by the number of phases k and the number of stages $\lceil \log_\eta W \rceil$ gives the stated bound. □

The Recursive Algorithm The sparse skeleton spanner algorithm forms the basis for our recursive approximate SSSP algorithm. The algorithm is divided into two stages. The purpose of the first stage is to compute a hierarchical structure of spanners $G_1, \ldots, G_T$ as follows. Let $G_0 := G$, and choose parameters $\alpha \geq 5$, $h := c\alpha$ for a sufficiently large constant c, $k := \log_\alpha n$, and constant $\eta > 1$. We construct the first sparse spanner G_1, which contains all nodes of G, by performing the distributed spanner algorithm of Baswana and Sen [BS07] in the local network with parameter k, where every node becomes responsible for all edges that it adds to the spanner throughout the algorithm's execution. By [BS07, Theorem 5.1], we obtain a $(2\log_\alpha n - 1)$-spanner in time $\mathcal{O}(\log^2_\alpha n)$. Furthermore, a closer inspection of the proof of [BS07, Theorem 4.1] shows that G_1 is in $\widetilde{\mathcal{O}}(\log_\alpha n \cdot n^{1/\log_\alpha n}) = \widetilde{\mathcal{O}}(\alpha)$-oriented form, w.h.p. Every other spanner G_i ($i \geq 2$) is constructed as an h-hop skeleton spanner of G_{i-1}, where every node in G_{i-1} joins the set M_i of marked nodes with probability $\min\{1, \log(n)/\alpha\}$ for $i = 2$ and with probability $1/\alpha$ for $i \geq 3$. To synchronize the executions of the iterative spanner constructions, we use Aggregate-and-Broadcast Algorithm. When for the first time a spanner G_{T+1} contains no nodes anymore, the first stage of the algorithm ends.

After the first stage has finished, in the second stage we simply perform the distributed Bellman-Ford algorithm with source s in the union of all recursively constructed spanners $H = \bigcup_{1 \leq j \leq T} G_i$ for $\mathcal{O}(\alpha \log_\alpha n)$ rounds. Finally, every node $v \in V$ chooses the minimum of all received distance values as its estimate $\tilde{d}(s, v)$ of $d(s, v)$. In the following, we show that H is a good spanner of the underlying graph G and that, moreover, between any two nodes of G, there is a path consisting of at most $\mathcal{O}(\alpha \log n)$ hops in H whose length gives a good approximation of the distance between the nodes in G. We first need a technical lemma.

Lemma 6.15 (Short Paths in G_i)**.** *Assume that P is a shortest path on G_1 between two nodes of G_1 that also has fewest hops. Further consider $i \geq 1$ and let $u, v \in M_i$ be two nodes on the path P such that $hop_{G_1}(u, v) = q$ for some $q \in [\gamma \cdot \alpha^{i-1}, \gamma \cdot \alpha^i]$. For a sufficiently large constant γ, u and v are connected in G_i by a path that consists of at most $\mathcal{O}(\alpha)$ hops and that has length at most $(2\eta k)^i \cdot d_{G_1}(u, v)$.*

Proof. We prove the lemma by induction on i. For $i = 1$, the statement holds directly as u and v are connected by a path in G_1 with hop-distance at most $\gamma \cdot \alpha = \mathcal{O}(\alpha)$ by assumption already.

We can therefore focus on the induction step and $i \geq 2$. Let P' be the subpath of P between u and v. Further, let $M_i^{(P')}$ be the set of nodes of M_i that are on path P'. Clearly, $M_i^{(P')}$ contains u and v. Note also that for all $i \geq 2$, nodes of V are sampled to be in M_i with probability $\log(n)/\alpha^{i-1}$. Because P' contains $q \geq \gamma\alpha^{i-1}$ hops, for a sufficiently large constant γ we have that $M_i^{(P')} = \Omega(\log n)$, w.h.p. Further, because q is upper bounded by $\gamma\alpha^i$, it also holds $M_i^{(P')} = O(\alpha \log n)$, w.h.p.

Our goal is to select a subset M_i' of $M_i^{(P')}$ of size $\mathcal{O}(\alpha)$ that contains u and v and such that for any two consecutive nodes x and y of M_i' on P', it holds that the subpath of P' connecting x and y contains between $\gamma\alpha^{i-1}/5$ and $\gamma\alpha^{i-1}$ hops in G_1. To see that this is always possible, we partition the path P' into arbitrary subpaths that all contain between $\gamma\alpha^{i-1}/5$ and $\gamma\alpha^{i-1}/4$ hops. Because P' contains $\kappa \cdot \gamma\alpha^{i-1}/5$ hops for some $\kappa \geq 5$, we can always partition it into $\lfloor \kappa \rfloor$ subpaths of the required range. We then select a maximal set of non-adjacent subpaths that contains the first and the last of the subpaths (the ones containing u and v). Since each node of P' is contained in M_i' with probability $\log n/\alpha^{i-1}$, for sufficiently large γ each of the subpaths contains at least one node of $M_i^{(P')}$, w.h.p. We add u and v and an arbitrary node from each other selected subpath to M_i', which gives a set M_i' with the required properties.

Because the hop-distance between any two consecutive nodes x and y of $M_i' \subset M_i$ in P' is between $\gamma\alpha^{i-1}/5 \geq \gamma\alpha^{i-2}$ and $\gamma\alpha^{i-1}$, we can apply the induction hypothesis to the subpath between x and y and conclude that x and y are connected in G_{i-1} by a path consisting of $\mathcal{O}(\alpha)$ hops and of total length at most $(2\eta k)^{i-1} \cdot d_{G_1}(x, y)$. Choose the constant c in the definition of $h = c\alpha$ sufficiently large such that this path contains at most h hops. By Lemma 6.12, x and y are therefore connected in G_i by a path consisting of at most two hops and of length $l_{xy} \leq (2\eta k) d_{h,G_{i-1}}(x, y)$. Note that the induction hypothesis then also yields that $d_{h,G_{i-1}}(x, y) \leq (2\eta k)^{i-1} d_{G_1}(x, y)$, which implies that $l_{xy} \leq (2\eta k)^i d_{G_1}(x, y)$. Since P is a shortest path, we can sum up the lengths of all these subpaths for all x and y and obtain a path between u and v in G_i that has $\mathcal{O}(\alpha)$ hops and length $(2\eta k)^i d_{G_1}(u, v)$, proving the lemma. $\square$

We next prove that in the union spanner graph H, there is a short path between any two nodes that consists of few hops and approximates distances in G_1 well.

Lemma 6.16 (Stretch of H)**.** *Let $u, v \in V$ be two nodes of G_1 and let P be a shortest path between u and v in G_1. Assume that P consists of q hops and assume that for the constant γ from Lemma 6.15, ξ is the smallest integer for which $q \leq \gamma\alpha^\xi$. Then graph H contains a path that consists of at most $\mathcal{O}(\alpha \log_\alpha n)$ hops and that has length at most $(2\eta k)^\xi d_{G_1}(u, v)$.*

Proof. We prove the lemma by induction on ξ. First note that for $\xi = 1$, the claim directly follows because $|P| = \mathcal{O}(\alpha)$. Let us therefore consider $\xi \geq 2$. Recall that the nodes in M_ξ are sampled with probability $\log(n)/\alpha^{\xi-1}$. Hence, w.h.p., for sufficiently large γ, every subpath of length at least $\gamma\alpha^{\xi-1}/3$ contains at least one node of M_ξ. Specifically, since $|P| > \gamma\alpha^{\xi-1}$, P contains at least one node of M_ξ.

Let x and y be the first and the last node of M_ξ on P when going along the path from u to v, and let P_1 be the subpath from u to x, P_2 be the subpath from x to y, and P_3 be the subpath from y to v. Note that by the above observation, $|P_1| \leq \gamma\alpha^{\xi-1}/3$ and $|P_3| \leq \gamma\alpha^{\xi-1}/3$ from v. We will first argue that for P_2, H contains a path that has at most $\mathcal{O}(\alpha)$ hops and length at most $(2\eta k)^\xi \cdot d_{G_1}(x, y)$.

First, note that since $|P| \leq \gamma\alpha^\xi$, we clearly also have that $|P'| \leq \gamma\alpha^\xi$. Let us first assume that P' contains at least $\gamma\alpha^{\xi-1}$ many hops. Then Lemma 6.15 implies that

G_ξ (and thus H) contains a path consisting of $\mathcal{O}(\alpha)$ nodes and of total length at most $(2\eta k)^\xi \cdot d_{G_1}(x, y)$. If otherwise $|P'| < \gamma\alpha^{\xi-1}$, we must have that $|P'|\gamma\alpha^{\xi-1}/3 \geq \gamma\alpha^{\xi-3}$, since P_1 and P_2 contain at most $\gamma\alpha^{\xi-1}/3$ hops, and since $|P| \geq \gamma\alpha^{\xi-1}$, P_2 must be larger. Since $M_\xi \subseteq M_{\xi-1}$, x and y must also be contained in $M_{\xi-1}$. In this case, we can apply Lemma 6.15 to argue that $G_{\xi-1}$ (and thus H) contains a path consisting of $\mathcal{O}(\alpha)$ nodes and of total length at most $(2\eta k)^{\xi-1} \cdot d_{G_1}(x, y)$.

It remains to consider the subpaths P_1 and P_2. For P_1, we can inductively argue that it can be divided into two subpaths, where the first subpath ends at a node w of $M_{\xi-l}$ for some $l \geq 1$, and the second subpath ends at x (which must also be contained in $M_{\xi-l}$). Whereas the second subpath contains an $\mathcal{O}(\alpha)$-hop path of length at most $(2\eta k)^{\xi-1} \cdot d_{G_1}(w, x)$ using the above argument, the first subpath needs to recursively be divided further. Whereas this recursion does not worsen the distance approximation factor, it does add an $\mathcal{O}(\log_\alpha n)$ factor to the number of hops of the resulting total path. The same argument can be applied to P_3, which proves the claim. □

We are now ready to prove Theorem 6.8.

Proof of Theorem 6.8. First, we show the approximation factor. Let $u, v \in V$ be two nodes. By Lemma 6.16 and since $T = O(\log_\alpha n)$, between any two nodes $u, v \in V$, the combined spanner H contains a path of length at most

$$(2\eta k)^T \cdot d_{G_1}(u, v) = O(k)^{\mathcal{O}(\log_\alpha n)} \cdot d_{G_1}(u, v),$$

and this path consists of at most $\mathcal{O}(\alpha \log_\alpha n)$ hops. The spanner G_1, which is computed by using the spanner algorithm of Baswana and Sen [BS07], also contains a path of at most $\mathcal{O}(k)$ hops and with distance stretch $\mathcal{O}(k)$ between any two nodes in V. Therefore, we can conclude that between any two nodes $u, v \in V$, the spanner H contains a path of length $(\log_\alpha n)^{\mathcal{O}(\log_\alpha n)} \cdot d_G(u, v)$, consisting of at most $\mathcal{O}(\alpha \log_\alpha n) = \widetilde{\mathcal{O}}(\alpha)$ hops. This particularly shows that by propagating distances from s for $\mathcal{O}(\alpha \log_\alpha n)$ rounds in H, every node $v \in V$ learns a distance estimate $\tilde{d}(s, v) \leq (\log_\alpha n)^{\mathcal{O}(\log_\alpha n)} \cdot d(s, v)$.

Finally, we show the runtime of our algorithm. In the first stage, constructing the first spanner G_1 takes time $\mathcal{O}(\log_\alpha^2 n)$, which follows from [BS07, Theorem 5.1] and our choice of $k = \log_\alpha n$. The resulting spanner is in $\widetilde{\mathcal{O}}(\alpha)$-oriented form, because the expected number of edges contributed to the spanner by a vertex in each of the $k = \log_\alpha n$ iterations of the algorithm is $n^{1/k} = \alpha$, and thus $\widetilde{\mathcal{O}}(\alpha)$, w.h.p. By our choice of k, every spanner G_i, $i \geq 2$, that is constructed using our sparse skeleton spanner algorithm is also in $\widetilde{\mathcal{O}}(\alpha)$-oriented form by Lemma 6.14. Therefore, constructing G_i takes time $\widetilde{\mathcal{O}}(\alpha |M|^{1/\log_\alpha n} \alpha) = \widetilde{\mathcal{O}}(\alpha^3)$. Furthermore, we have that $T = O(\log_\alpha n)$, which follows from the fact that the probability for a node to be in spanner G_{i+1} is $\log n/\alpha^i$. Thus, the first stage takes time $\widetilde{\mathcal{O}}(\alpha^3)$, w.h.p.

Now consider the second stage. As every spanner is in $\widetilde{\mathcal{O}}(\alpha)$-oriented form, in H every node is responsible for $\widetilde{\mathcal{O}}(\alpha)$ edges. Therefore, the broadcast trees necessary to perform multi-aggregations can be set up in time $\widetilde{\mathcal{O}}(\alpha)$, w.h.p., and every round of the Bellman-Ford computation can be realized by performing multi-aggregations in time $\widetilde{\mathcal{O}}(\alpha)$ as described in Section 4.4.2, for example. Since we only perform $\widetilde{\mathcal{O}}(\alpha)$

rounds of Bellman-Ford until every node has learned its correct distance, the second stage takes time $\widetilde{\mathcal{O}}(\alpha^2)$, w.h.p. $\square$

6.3. Outlook

Whereas the APSP Problem can be computed in asymptotically optimal time (up to polylogarithmic factors), and also the Diameter Problem is almost settled to the lower bound, no formal lower bound for the SSSP Problem is known so far. Clearly, coming up with such a lower bound would give valuable insight into the complexity of the problem. At this point, there is no reason to assume that any of the known upper bounds for SSSP are tight. It would be worthwhile to investigate whether the $\widetilde{\mathcal{O}}(\sqrt{\mathsf{SPD}})$-time algorithm of this chapter, or the $\widetilde{\mathcal{O}}(n^{1/3})$-time solution by Censor-Hillel et al. [CLP20] can be improved. Furthermore, studying variants of the SSSP Problem, such as the k-SSP or the (h,k)-SSP Problem, may yield interesting results.

Regarding approximate solutions, an improvement of our $\widetilde{\mathcal{O}}(n^{1/3})$-time algorithm for $(1+\varepsilon)$-approximate SSSP could be attainable. One of the most intriguing question is whether constant factor, in particular $(1+\varepsilon)$, approximations of SSSP can be computed in subpolynomial time.

Part II.

Hybrid Programmable Matter

7 Shape Formation in Hybrid Programmable Matter

Our first chapter on hybrid programmable matter studies the problem of *shape formation*. Arguably, transforming an initial structure into a specific shape is among the most important problems for programmable matter. In addition to the potential applications of shape formation mentioned in Chapter 1, a particularly interesting usage lies in the construction of *space facilities* [Nie+20; Abd+20]. Due to the inhospitable nature of space, and the exceptional costs of erecting and maintaining facilities by outside intervention, scalable and simple autonomous systems are of invaluable utility there. In recent years, there have been significant advances both in the development of light-weight lattice structures and in autonomous robotic systems to manipulate these structures. To make these systems cost-efficient, scalable, and robust, the robots and lattice structures need to be as simple, and the algorithms as efficient as possible.

In this chapter, we establish some theoretical foundations towards this goal. Specifically, we present a simple two-dimensional model for hybrid programmable matter. In this model, we show that already a *single* robot with constant-size memory is able to solve simple shape formation problems. Our ultimate goal is to investigate how multiple robots can cooperate to speed up the process of shape formation. To that end, we present a distributed shape formation algorithm that constructs a line using multiple robots and evaluate the performance of the algorithm experimentally.

Underlying Publication The chapter is based on the following publication.

> R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, C. Scheideler, and T. Strothmann. “Forming Tile Shapes with Simple Robots”. In: *Proceedings of DNA Computing and Molecular Programming (DNA).* 2018, pp. 122–138 [Gmy+18c]

An extended version of the paper has also been published in the *Natural Computing* journal [Gmy+20].

> R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, C. Scheideler, and T. Strothmann. “Forming tile shapes with simple robots”. In: *Natural Computing* 19.2 (2020), pp. 375–390 [Gmy+20]

Some preliminary results of the publication have been presented at the EuroCG 2017 Workshop [Gmy+17b].

Model Before we describe our results and give an outline of the chapter, we formally introduce the model upon which this and the following chapter are based.

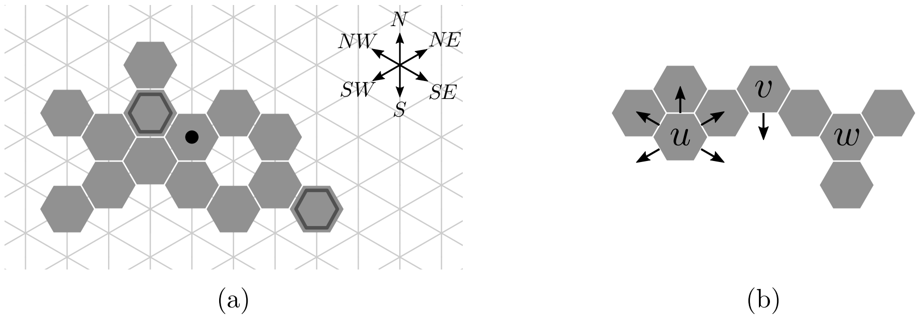


Figure 7.1.: (a) A connected set of tiles positioned on the triangular lattice. The black dots indicate the position of the robot, and gray contours indicate tiles that are marked by a pebble. (b) Possible movements of tiles u, v, and w. Tile w cannot be moved anywhere without violating connectivity.

We assume that a single *active* agent (a *robot*) operates on a finite set of n *passive* hexagonal *tiles.* Each tile occupies exactly one node of the infinite triangular lattice $G = (V, E)$ (see Figure 7.1a). A *configuration* (T, p) consists of a set $T \subset V$ of all nodes occupied by tiles, and the robot's position $p \in V$. The *diameter* of a configuration is defined as the maximal length of a shortest node path between any two occupied nodes of the triangular lattice. Note that every node $u \in V$ is adjacent to six neighbors, and, as indicated in the figure, we describe the relative positions of adjacent nodes by the six compass directions *N*, *NE*, *SE*, *S*, *SW* and *NW.*

Whereas tiles cannot perform any computation nor move on their own, the robot may change its position and carry a tile, thereby modify a configuration. The robot must stand on or be adjacent to a node occupied by a tile. Additionally, if the robot does not carry a tile, we require the subgraph of G induced by T to be connected; otherwise, the subgraph induced by $T \cup \{p\}$ must be connected. In a scenario where a tile structure swims in a liquid or floats in space, for example, this restriction prevents the robot or parts of the tile structure from drifting apart. Some examples of possible tile moving steps are shown in Figure 7.1b.

Additionally, the robot may carry a set of $k \geq 0$ indistinguishable *pebbles.* Pebbles can be placed on tiles in order to *mark* them. A tile can be marked by at most one pebble (see Figure 7.1a).

Initially, we assume that the robot stands on a tile of a connected structure and does not carry a tile. Furthermore, the robot carries all of its pebbles, and no tile is marked by a pebble already.

The robot operates in rounds of `Look-Compute-Move` cycles. In the `Look` phase of a round, the robot can observe its node p and the six neighbors of that node. For each of these nodes, it can determine whether the node is occupied or not, and if it is occupied, whether the tile is marked by a pebble. In the `Compute` phase, the robot may change its internal state and determines its next move according to the observed information. In the `Move` phase, the robot can perform at most one of the following actions:

(1) Lift a tile from p, if $p \in T$,

(2) place a tile it is carrying at p, if $p \notin T$,

(3) pick up a pebble, if $p \in T$ and the tile at p is marked by a pebble,

(4) place a pebble, if $p \in T$, the tile at p is not marked by a pebble, and the robot still has a pebble at its disposal,

(5) or move to an adjacent node.

Note that whereas the robot can carry at most one tile, it may carry multiple pebbles.

Formally, we model the execution of an algorithm by the robot as transitions of a *deterministic finite automaton* $(Q, \Sigma, \delta, q_0, F)$. Q is a constant-size set that contains all of the robot's possible states. $\Sigma = \{0, 1, 2\}^7$ represents the set of possible views of the robot: The first digit of an element in Σ indicates whether p is empty (0), occupied by a tile (1), or occupied by a tile on which a pebble is placed (2). The other six digits indicate, in order, the states of the adjacent nodes in direction *N*, *NE*, *SE*, *S*, *SW*, and *NW* from p. In each round, the robot executes one transition of a transition function δ, which is defined as

$$\begin{aligned} \delta : \quad & Q \times \Sigma \times \{0,1\} \times \{0, \dots, k\} \\ & \to Q \times \{\mathit{none}, \mathit{liftT}, \mathit{placeT}, \mathit{pickP}, \mathit{placeP}, \mathit{move}_d\}. \end{aligned}$$

The input of δ is the current state and view of the robot, a bit that indicates whether the robot carries a tile, and a value that gives the number of pebbles the robot currently carries. As an output, the transition function determines the robot's next state as well as one of the following actions: do nothing (*none*), lift a tile (liftT), place a tile (*placeT*), pick up a pebble (*pickP*), place a pebble (*placeP*), or move in some direction d of the six cardinal directions (move_d). $q_0 \in Q$ is the initial state of the robot, and $F \subseteq Q$ contains all final states. If in some round the robot is in a final state, it will not perform any further state transition; in this case we say the robot has *terminated*.

Note that we use the above definition to formally argue about the robot's capabilities and limitations. However, we will present our algorithms from a higher level by describing their behavior textually and through pseudocode. We remark that all our algorithms can easily be transformed into actual state machines. Further, note that even though we describe the algorithms as if the robot knew its global orientation, we do not actually require the robot to have a compass. For the algorithms presented in this and the following chapter, it is enough for the robot to be able to maintain its orientation with respect to its initial orientation.

Contribution and Outline In this chapter, we mainly focus on the *Triangle Formation Problem* with a single robot, in which the goal is to transform the set of all tiles into a triangular form. In Section 7.1, we begin by pointing out one of the limitations of our model: It is in general impossible for a single robot to find a tile that can be removed without disconnecting the tile structure. We contrast this result by showing that having a single *pebble* already suffices to solve this problem.

We then show how to construct *intermediate structures* by using simple tile movements that allow for easy navigation and tile removal in Section 7.2. More specifically, we present three intermediate structures. The simplest among them is a *line* structure; it can be constructed in $\mathcal{O}(n^2)$ rounds (Section 7.2.1). The second structure we introduce is a *block*. It has diameter $\mathcal{O}(D)$, where D is the structure's initial diameter, and can often be constructed more efficiently than the line, namely in $\mathcal{O}(nD)$ rounds (Section 7.2.2). Finally, we describe a *tree* structure, which, in contrast to the previous structures, can be built completely inside the convex hull of the original tile set in $\mathcal{O}(n^2)$ rounds (Section 7.2.3). Using the block structure as an example, we argue that each of these intermediate structures can be transformed into a triangle by performing an additional $\mathcal{O}(nD')$ rounds (D' being the intermediate structure's diameter). This leads to an algorithm that solves the Triangle Formation Problem in time $\mathcal{O}(nD)$ for initial diameter D.

In Section 7.4, we finally discuss how the line algorithm can be transferred to the multi-robot case. We provide some first simulation results showing that a small number of robots can speed up line formation by a significant amount. As the number of robots increases, we observe the anticipated decline in speedup. We then describe how a triangle can be built from a line in a distributed manner.

Related and Subsequent Work There is a number of approaches to shape formation in the literature that use agents that fall somewhere in the spectrum between passive and active. As mentioned in Chapter 1, *tile-based self-assembly* [Pat14] uses passive tiles that bond to each other to form shapes. A variant of *population protocols* [Ang+06] proposed by Michail and Spirakis [MS16] uses agents that are partly passive (i.e., they cannot control their movement) and partly active (i.e., upon meeting one another, they can perform a computation and decide whether they want to form a bond). Finally, the *Amoebot model* [Der+14], the *nubot model* [Woo+13], and the two-dimensional modular robotic model [Hur+15] use agents that are completely active in that they can compute and control their movement. Shape formation has also been investigated for practical modular robotic systems (see, e.g., [MKK94; Tom+99]). Here, the robots typically have much greater computational capabilities than in our model.

We remark that some of the above models are more powerful than our model and could therefore easily simulate our algorithms. For example, in the Amoebot model a set of n active agents could form the initial tile structure and simulate movements of the active agent by transferring its role from one agent to another. As every agent is able to move in that model, modifications of the tile structure can, in principle, also be simulated, although coming up with an actual simulation framework might still be difficult. In contrast, the simplicity of our model allows us to focus on the question whether already a *single* active agent with the power to manipulate the structure of passive agents suffices for complex tasks such as shape formation.

Recently, Fekete et al. [Fek+21] studied a simplified version of our model, which considers square tiles and allows the robot to move on unoccupied nodes and create and destroy tiles at will. Based on this model, they solve geometric tasks such as computing a bounding box, counting tiles, or scaling and rotating the tile structure. To do so, they exploit the capability of the robot to essentially construct a Turing machine using tiles. Furthermore, the algorithm does not preserve connectivity. However, using two robots (or, alternatively, one robot and a pebble), the algorithms can be adapted to maintain connectivity [Nie+20; Fek+20].

When arguing about a robot traversing a tile structure without actually moving tiles, our model reduces to an instance of the ubiquitous *agents on graphs* model. Research in this model covers many problems, including *Gathering* and *Rendezvous* (e.g., [Pel12]), *Intruder Caption* and *Graph Searching* (e.g., [BN11; FT08]), *Graph Exploration* (e.g., [Das13]), or *Black Hole Search* (e.g., [Mar12]). Other approaches allow agents to move tiles (e.g., [Dem+03; TM08]), but these focus on computational complexity issues or agents that are more powerful than finite automata.

7.1. Finding Safely Removable Tiles

In a naive approach to shape formation, the robot could iteratively search for a tile that can be fully removed from the structure without disconnecting the tile structure (a *safely removable tile*) and then move that tile to some position such that the shape under construction is extended. More formally, a safely removable tile is a tile that does not occupy a *cut vertex* v of the subgraph H of G induced by the nodes of T (i.e., a node whose removal from H does not increase the number of components in H). Since H is finite, not every node of H can be a cut vertex; therefore, there always is a safely removable tile. However, the following theorem shows that, in general, a single robot cannot *decide* whether a tile is safely removable, which makes this naive approach infeasible.

Theorem 7.1 (Safely Removable Tiles)**.** *There does not exist a deterministic finite automaton $\mathcal{A}$ so that if the robot executes $\mathcal{A}$ on any configuration starting on an occupied node and without carrying a tile, it (1) never performs a tile lift, (2) terminates on a safely removable tile.*

Proof. Suppose that there is such an automaton $\mathcal{A}$, and let $s = |Q|$. We consider the configuration H_ℓ in which the tiles form a hollow hexagon of side length ℓ, and place the robot on the southernmost tile of the hexagon as depicted in Figure 7.2a (we call this node the *southern vertex* of H_ℓ). We define the set of *border nodes* to be all vertices (i.e., the corners) of the hexagon, all empty nodes inside the hexagon that are adjacent to a vertex, and all empty nodes outside the hexagon whose only neighbor is a vertex (see Figure 7.2a). Consider the finite sequence of *system states* $(S_1, S_2, \ldots, S_T)$ through which the robot progresses while executing $\mathcal{A}$, where $S_i = (p_i, q_i)$ contains the robots position p_i and state q_i before executing round i. S_1 corresponds to the initial system state, and S_T is the first system state for which $q_T \in F$. We partition this sequence into *phases*, where we define a new phase to start whenever the robot visits a border node (i.e., for every phase $(S_i, \ldots, S_k)$, p_i is a border node, and for all j, $i < j \leq k$, p_j is not a border node).

Note that since there are at most 18 border nodes, there can be at most $18s$ phases: Otherwise, there must be two phases that begin with system states S_i, S_j, respectively, such that $S_i = S_j$ (w.l.o.g., let $i < j$). Since the tile structure is never altered by the robot, $S_{i+1} = S_{j+1}$, and, inductively, $S_{i+k} = S_{j+k}$ for all $k \geq 0$, which

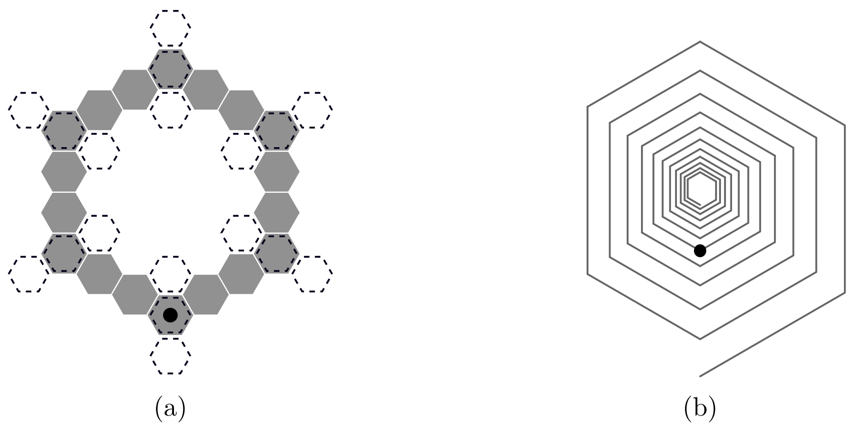


Figure 7.2.: (a) The hollow hexagon of side length $\ell = 4$. The border nodes of the hexagon are marked by dashed frames. (b) An example of the tile structure T. In both figures, the black dot indicates the initial position of the robot.

implies an infinite loop and contradicts the assumption that the sequence of system states is finite.

The way the robot traverses the hexagon depends on the side length ℓ. We define the *traversal sequence* associated with ℓ as the sequence $(S_1, S_2, \ldots, S_k)$ of all system states a phase begins with when $\mathcal{A}$ is executed on H_ℓ (i.e., S_i is the first system state of phase i, and k is the total number of phases). Note that a traversal sequence may be of length 1, i.e., if the robot never visits a border node except for its initial position. Since the algorithm takes at most $18s$ phases to terminate (for any choice of ℓ), there can only be at most $(18s)^{18s}$ distinct traversal sequences for different choices of ℓ. Hence, there is a finite number of traversal sequences and an infinite number of side lengths, which, according to the pigeonhole principle, implies that there must be an infinite set L of side lengths corresponding to the same traversal sequence.

Based on this observation, we now define a tile structure $\mathcal{T}$ for which the robot terminates on a tile that is not safely removable. This tile structure essentially consists of a spiral as depicted in Fig. 7.2b. We start at an arbitrary node of the triangular lattice and construct an outward spiral consisting of $72s$ line segments of tiles. The first line segment of the spiral goes NW and each subsequent line segment takes a $60°$ clockwise turn. The lengths of the line segments are chosen from L in such a way that the smallest side length ℓ_{min} is larger than $s + 2$, and such that the segments stay well-separated. This is possible since L is an infinite set and therefore we can always choose sufficiently large segment lengths. We initially place the robot at the last tile of the $(36s)$-th line segment, which is a tile with neighbors at NW and NE.

It remains to show that the algorithm fails to find a tile that can be safely removed when being executed on $\mathcal{T}$. As above, we subdivide the execution of the algorithm into phases, where we define a new phase to start whenever the robot visits a border node of the spiral (which, analogous to the definition for the hexagon, we define as the three nodes at each turn of the spiral). Using induction on the phases, we show that the robot traverses $\mathcal{T}$ in a way that corresponds to the traversal sequence associated with the side lengths in L.

More specifically, we show that the i-th border node visited by the robot on $\mathcal{T}$ (1) has the exact same neighborhood as the i-th border node visited by the robot in a hexagon H_ℓ for all $\ell \in L$, and (2) is visited in the same state. This initially holds as the robot is placed on a tile with only *NW* and *NE* neighbors in both structures and starts in the initial state. For $1 < i \leq 18s$, w.l.o.g., assume that the i-th border node visited in $\mathcal{T}$ is occupied and has a tile at *NW* and *NE* (all the other cases are analogous), and the robot is in state q (note that the robot cannot have reached either end of the spiral after having visited fewer than $36s$ border nodes). Let ℓ_{NE} be the length of the line segment in direction *NE* from the robot's current position, and let ℓ_{NW} be the length of the line segment in direction *NW*. By the induction hypothesis, we have that the i-th border node v visited on H_ℓ is the the southern vertex of the hexagon for all $\ell \in L$, and the robot is in state q when visiting the i-th border node.

W.l.o.g, assume that the next border node visited by the robot on H_ℓ is not adjacent to the south-east vertex of H_ℓ (i.e., the robot does not follow the hexagon in direction *NE*). Then, in any hexagon H_ℓ, the robot will never move away from v in direction *NE* by more than s steps. Since the length of the line segment in direction *NE* is larger than $s+2$, the robot would otherwise visit two nodes with the same neighborhood in the same state, which would cause a repetition that leads the robot to a border node at the south-east vertex, which contradicts our assumption. Therefore, for every node visited by the robot on $H_{\ell_{NW}}$ before visiting a border node in direction *NE*, the robot visits a node with the *same* neighborhood on $\mathcal{T}$. Therefore, the next border node visited by the robot on $\mathcal{T}$ has the same neighborhood as the next border node visited by the robot on $H_{\ell_{NW}}$, and is visited in the same state. Since all $\ell \in L$ exhibit the same traversal sequence, we conclude the induction.

Therefore, the $(18s)$-th visited border node on $\mathcal{T}$ corresponds to the last border node visited by the robot on H_ℓ for all ℓ, from where the robot will not move farther away than by $s < \ell_{min} - 2$ steps before terminating on a tile (otherwise, there would again be a repetition leading the robot to a border node). However, since the robot has only visited at most $18s$ border nodes, it cannot have reached either end of the spiral, and thus terminates on a tile that is not safely removable. This directly contradicts the assumption that the automaton works correctly and therefore shows that there is no such automaton. $\square$

In contrast, the problem can be solved by equipping the robot with a single pebble. We first describe how the robot can use a single pebble to detect whether a given tile is safely removable. Let S be a maximal set of connected empty nodes given some tile configuration. If S is finite, then it is a *hole*; otherwise, it is the infinite set of empty nodes around the structure. We refer to the subset of S adjacent to occupied nodes as the *boundary* of S. Any tile t can be adjacent to at most three different boundaries (see Figure 7.3a). We define the *outline* of a boundary as the set of its adjacent occupied nodes. For a tile t, consider the subgraph H of G induced by the empty nodes adjacent to the node of t. H has at most three components, which we

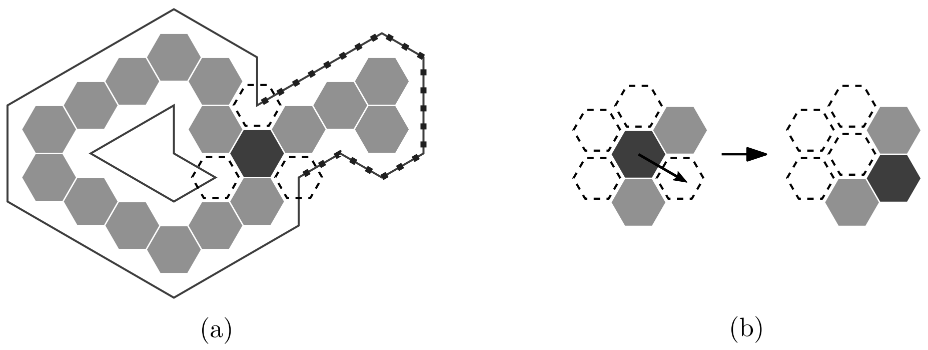


Figure 7.3.: (a) A tile t (black) that is not safely removable with its three empty regions (dashed outlines). t is adjacent to two boundaries (black lines). The dotted path could be extended to a cycle if t was removed. (b) A local tile movement that preserves connectivity.

call t's *empty regions* (see the black tile in Figure 7.3a). Note that every region is contained in some boundary that we refer to as its *corresponding boundary*. In the example of a line segment, all empty regions correspond to the same boundary.

Lemma 7.2. *A tile t is safely removable if and only if all of its empty regions correspond to different boundaries.*

Proof. First, assume that t has at least two empty regions that belong to the same boundary (e.g., the outer boundary in Figure 7.3a). Consider a path P of empty nodes along the boundary that connects the two regions of t (the dotted path in Figure 7.3a). After the removal of t, we can extend P to a cycle C using the remaining empty node of t. C consists of empty nodes and surrounds a set of tiles A. However, as t had at least two empty regions, there must remain a tile t' adjacent to t that is not connected to any tile of A. Therefore, the configuration cannot be connected anymore.

Now assume all empty regions of t belong to separate boundaries, and consider two different outlines containing t. The empty regions that are part of the corresponding boundaries are connected via tiles in t's neighborhood. Hence, any two outlines containing t are connected via tiles adjacent to t and thus remain connected after removing t. Thus, t is safely removable. □

Now we are ready to give our automaton to find a safely removable tile, which, for simplicity, we describe as an algorithm for the robot. To search for a safely removable tile, the robot first walks N, NW, and SW (in that precedence) until it reaches a *locally northwesternmost tile* t (i.e., a tile with no neighbors at N, NW, and SW). The empty node NW of t belongs to a boundary whose outline O contains t. It can easily be seen that O contains a safely removable tile. To find it, the robot traverses O in clockwise order and checks each tile t' of O separately using the following procedure. First, it places the pebble on t'. Then, it traverses each boundary adjacent to t' and verifies whether it returns to t' within the same region, in which case all empty regions belong to separate boundaries. Together with Lemma 7.2, we conclude the following theorem.

Theorem 7.3 (Safely Removable Tiles with a Pebble)**.** *A single robot can find a safely removable tile in $\mathcal{O}(n^2)$ rounds with the help of a single pebble.*

7.2. Forming an Intermediate Structure

Although the robot cannot always find a safely removable tile (unless being equipped with a pebble), it can always perform local tile movements that preserve connectivity. For example, when a tile t only has adjacent tiles at *NE* and *S*, removing t from the structure may potentially disconnect the system (see Figure 7.3b). However, in this case we can still pickup t, move one step *SE* (where connectivity is preserved through the carried tile), and reconnect the shape by placing it there. In this section, we show how to construct *intermediate structures* by performing such movements. In the resulting structures the robot can easily navigate and move tiles without possibly violating connectivity. Therefore, it can easily disassemble such a structure and rearrange its tiles into the desired shape.

We aim to construct *simply connected* intermediate structures (i.e., structures without holes), as removability of a tile can easily be determined locally in such a structure: A tile is safely removable if and only if it only has one empty region. Such a tile can always easily be found in a simply connected structure. Note that although in the presented intermediate structures it is easy to determine a location where an arbitrarily large shape can be built, a robot may not always be able to find such a location in *any* simply connected structure.

We show how to construct three different intermediate structures. As a first simple example, we demonstrate how to construct a *line* in time $\mathcal{O}(n^2)$, which is a sequence of connected tiles from north to south. Clearly, the main drawback of this algorithm is that tiles might need to be moved by a distance linear in n. Our second algorithm avoids this pitfall by building a structure called a *block* in time $\mathcal{O}(nD)$ with initial diameter D. The algorithm further ensures that no tile is moved farther than by a distance of D. The last and most complex algorithm builds a simply connected structure called a *tree* in time $\mathcal{O}(n^2)$. The main advantage of this solution is that no tile is ever placed outside of the *convex hull* of the initial configuration, which is the convex hull of the corresponding set of hexagonal tiles in the Euclidean plane.

7.2.1. Forming a Line

To construct a line, the robot first moves S as far as possible, i.e., as long as there is a tile in direction S. Then, it alternates between a *tile searching phase*, in which it moves NW, SW, and N (in that precedence) until there is no longer a tile in any of these directions, and a *tile moving phase*, in which it lifts the tile, moves one step *SE*, moves S until it reaches an empty node, and then places the tile. The line is complete once the robot does not encounter any adjacent tiles to the east or west in the tile searching phase. Figure 7.4 shows the first several steps of this algorithm.

Theorem 7.4 (Line Formation)**.** *A single robot can transform any connected tile configuration into a line and terminate after $\mathcal{O}(n^2)$ rounds.*

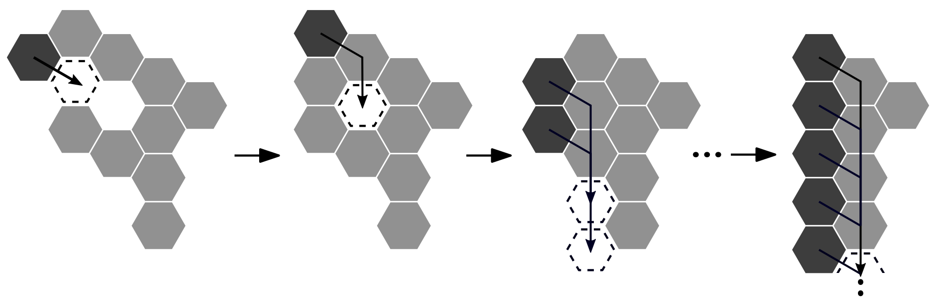


Figure 7.4.: First several steps of line formation. The black tiles are moved to the positions marked by dashed outlines.

Proof. We define a *column* as a maximal sequence of connected tiles from N to S. First, note that the robot always finds a locally northwesternmost tile in the tile searching phase. Furthermore, each tile movement always preserves connectivity. To prove that a line is eventually built, we observe that the tile searching phase does not terminate in the northernmost tile of a locally easternmost column as long as there is more than one column in the tile configuration. This follows from the fact that the tile searching phase always starts at the southernmost tile of a column, and that preference is given to the NW and SW directions when searching. Therefore, if a line has not yet been constructed, and the robot stops at some locally northwesternmost tile, there must be tiles east of that position. This observation specifically implies that no tile is ever moved east of the initially *globally* easternmost column, unless the structure already forms a line. Otherwise, the robot will at some point place a tile east of that column for the first time, at which point the column from which the tile is taken must be locally easternmost, contradicting the statement above.

To conclude the correctness of the algorithm, we argue that the robot terminates when a line is formed for the first time. In case that the structure initially forms a line already, the robot completely traverses the line from south to north in the first tile searching phase, finds no tile to the east or west, and terminates. Otherwise, the structure eventually becomes a line after the robot has placed the line's southernmost tile, in which case the robot terminates after the next tile searching phase.

Finally, we show that the algorithm takes $\mathcal{O}(n^2)$ rounds. The first steps of moving south take $\mathcal{O}(n)$ rounds. For the two phases, we first bound the number of times the robot moves a tile by one step in the tile moving phase. By the above observations, the tiles of an easternmost column in the initial tile configuration are never moved. Since furthermore the initial tile configuration is connected, and tiles are exclusively moved SE and S, each tile is moved at most $2n$ steps. Therefore, $\mathrm{s}_{\text{move}} = \mathcal{O}(n^2)$ move steps are performed in total.

Now, consider the tile searching phase. We assign coordinates to each node, where the x-coordinate grows from west to east and the y-coordinate grows from north to south. More precisely, the changes of the (x, y) coordinates for movements in the six cardinal directions are

$$\begin{aligned}
&N = (0, -1),\ NW = (-1, -1/2),\ SW = (-1, 1/2),\\
&SE = (1, 1/2),\ S = (0, 1),\ NE = (1, -1/2).
\end{aligned}$$

Note that whereas the sum of the coordinates of the robot increases at every step in the tile moving phase, it decreases at every step in the tile searching phase. More specifically, in each step of the tile moving phase, the sum of the coordinates of the robot increases by at most 3/2, and in every step of the tile searching phase, the sum of the coordinates decreases by at least 1/2. Thus, the total number of steps in the tile searching phase can be bounded by

$$\mathrm{s}_{\mathrm{search}} < 2 \cdot \left((3/2) \cdot \mathrm{s}_{\mathrm{move}} + s_{\mathrm{south}} + (x_0 + y_0) - \min_i (x_i + y_i) \right) ,$$

where $\mathrm{s}_{\mathrm{south}}$ is the number of steps the robot initially takes to go south, (x_0, y_0) denotes the robot's initial coordinates, and the value $\min_i(x_i + y_i)$ is taken over all possible placements of all tiles. As the initial tile configuration is connected, $\mathrm{s}_{\mathrm{south}} \leq n$, $(x_0 + y_0) - \min_i(x_i + y_i) = \mathcal{O}(n)$, and $\mathrm{s}_{\mathrm{move}} = \mathcal{O}(n^2)$. Therefore, the total number of search steps is $\mathcal{O}(n^2)$. Since each move and search step takes $\mathcal{O}(1)$ rounds, the total number of rounds is $\mathcal{O}(n^2)$. □

It is not hard to see that $\Omega(n^2)$ rounds are necessary to rearrange an arbitrary initial tile configuration into a line by a single robot. If starting from an initial configuration with diameter $\mathcal{O}(\sqrt{n})$, for example, a constant fraction of the tiles has to be moved by a distance linear in n. Therefore, a total of $\Omega(n^2)$ move steps are required to build a line.

7.2.2. Forming a Block

Although a line can be constructed efficiently, its linear diameter might make it an undesirable intermediate structure. In fact, if both the initial diameter and the diameter of the desired shape are small, moving tiles by a linear distance to construct an intermediate structure seems to be an excessive effort. Therefore, we introduce another intermediate structure, which is called a *block*: In a block, all tiles except those farthest to the west have an adjacent tile to the northwest. Therefore, a block has only one westernmost column, and every *row*, which we define as a maximal sequence of connected tiles from *NW* to *SE*, begins with a tile from that column (see right picture in Figure 7.5). Clearly, a line is a special case of a block that only consists of one column. Our algorithm builds a block in $\mathcal{O}(nD)$ time and does not move any tile farther than by a distance D, where D is the diameter of the initial structure. An example of the transformation of an initial structure into a block is shown in Figure 7.5.

We present the algorithm in two steps. First, we describe a non-halting algorithm by giving simple tile moving rules similar to the rules of the line construction algorithm. The algorithm will eventually build a block structure. We then extend the algorithm with additional checks to detect whether a block structure has been built.

As in the line algorithm, the robot alternates between a searching and a moving phase: It first searches for a locally northwesternmost tile by repeatedly moving *NW*, *SW*, or *N* (in that precedence). The robot then picks up the tile, moves *SE* until it reaches an empty node, and places the tile there. Note that at this point, the algorithm differs from the line algorithm only in that it skips the initial phase

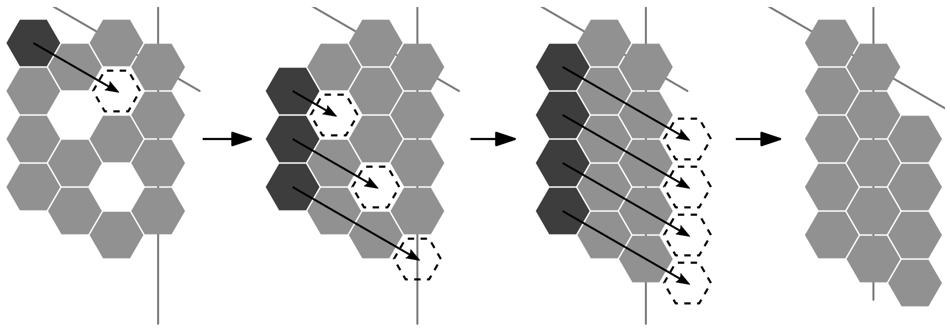


Figure 7.5.: Transformation of an initial structure into a block. The gray lines indicate some fixed x- and y-coordinates for reference.

of moving south, and that it places each tile at the first empty *SE* position instead of moving only *one* step *SE* and then placing the tile at the first southern position.

We show the correctness of this simple algorithm through a sequence of lemmas. As in the proof of Theorem 7.4, we assign coordinates to each node, where the x-coordinate grows from west to east and the y-coordinate grows from north to south. For the following three lemmas, let 0 be the maximum x-coordinate of all tiles in the initial tile configuration, i.e., the x-coordinates of the easternmost tiles are 0 and all others have negative x-coordinates.

Lemma 7.5. *During the algorithm's execution, any two tiles with x-coordinate* 0 *are connected via a simple path of tiles whose x-coordinates are at most* 0*.*

Proof. The claim initially holds. Let P be the simple path connecting two tiles u and v with x-coordinate equal to 0. We show that after the robot has moved any other tile $t \neq u, v$, there remains a path between u and v. If t has x-coordinate less than 0, placing the tile at the first empty position *SE* will maintain a simple path with x-coordinate at most 0. If t has x-coordinate equal to 0, then t cannot lie on P, as t does not have adjacent tiles at *N*, *NW*, and *SW*. Thus, moving t does not affect the path. □

Lemma 7.6. *If there is a tile with x-coordinate* 0*, and the robot picks up a tile at some node v with x-coordinate x_v, then $x_v \leq 0$.*

Proof. The node of the first tile the robot picks up has x-coordinate at most 0. Now assume that afterwards the robot picks up a tile at some node v of the triangular lattice with x-coordinate x_v. If there is a tile at the southern neighbor of v, then the next tile the robot will lift has x-coordinate at most x_v.

This implies that in order for the robot to lift the first tile t_2 with x-coordinate greater than 0, it has to have previously lifted a tile t_1 at 0 with no southern neighbor. At that point, t_1 also had no adjacent tile at *SW*, *NW*, and *N*. Therefore, it could not have been connected to any other tile at 0 via a path of tiles with x-coordinate at most 0. Thus, by Lemma 7.5, t_1 was the *only* tile at 0 when it was lifted. Therefore, there is no tile with x-coordinate 0 when t_2 is lifted. □

Note that in the next lemma we do not yet assume that the algorithm will terminate when a block structure has been built, but only show that a block will eventually be built.

Lemma 7.7 (Block Correctness)**.** *Let the maximum x-coordinate of the tiles in the initial tile structure be* 0*. Then the algorithm rearranges the tiles into a block, in which the westernmost column of tiles has x-coordinate* 1*, in $\mathcal{O}(nD)$ rounds.*

Proof. We first show the correctness of the algorithm. First, note that the robot always finds a tile to move. By Lemma 7.6, the robot will repeatedly pick tiles with x-coordinate at most 0 until there is no such tile anymore. At this point, every tile with x-coordinate at least 2 has a neighbor at *NW*. This is due to the fact that each such tile must have had a *NW* neighbor at the time of its placement, and by Lemma 7.6 none of these tiles has been moved yet. Therefore, the tiles are arranged as a block in which the westernmost tiles have x-coordinate at most 1.

We now turn to the runtime of the algorithm, which we prove similarly to Theorem 7.4. It is easy to see that each tile is moved for at most $2D$ steps until the block is established, which implies that at most $\mathcal{O}(nD)$ move steps are performed in total. Note that each time a tile is moved, the sum of the robot's coordinates increases by $3/2$. On the other hand, each search step decreases this sum by at least $1/2$. Using the same argument as in Theorem 7.4, the total number of search steps is therefore bounded by $\mathcal{O}(nD)$. Since each step is performed within a constant number of rounds, the total number of rounds until a block is built is $\mathcal{O}(nD)$. □

Next, we show how the robot can terminate once a block has been successfully built by performing a series of tests alongside the algorithm's execution. Note that according to the above algorithm the robot will move each tile of the westernmost column of a finished block, starting with the northernmost tile, placing each at the first empty position *SE* of it. Thereby, the robot can detect that a block has been built by verifying the following conditions: (1) after placing a tile, the robot performs at most one *SW* movement before it lifts the next tile, (2) while moving a tile t, the robot does not traverse a node (except for the position at which t was lifted) that has a neighbor at *NE*, but not at *N*, or a neighbor at *S*, but not at *SW*, (3) the robot never places a tile at a node that has a neighbor at *SE*. A test verifying the above conditions is initiated whenever the robot picks a tile that does not have a *NE* neighbor. If thereafter any of the above conditions gets violated, the test is aborted. If otherwise the robot lifts and places the southernmost tile of a column without having encountered any violation up to this point, the algorithm terminates.

Theorem 7.8 (Block Formation)**.** *A single robot can transform any connected tile configuration of diameter D into a block and terminate after $\mathcal{O}(nD)$ rounds.*

Proof. By Lemma 7.7, the robot builds a block within $\mathcal{O}(nD)$ rounds. Assume the robot lifts a tile t that does not have a *NE* neighbor and initiates the test sequence. If the structure is a block already, the robot will move its westernmost tiles as described above and, after moving the last tile of the westernmost column, the test series finishes without any violation.

Now assume the structure is not a block at the time t is lifted by the robot. We show that in this case the test series fails. If a tile s of the column below t has a neighbor at *SW*, the test series will fail at the latest when the robot moves the northern neighbor s' of s and afterwards takes at least two steps *SW* to reach s', thereby violating Condition 1. If otherwise no such tile exists, i.e., no tile in the column below t has a neighbor at *SW*, then there must be a tile s farther east than t that has no neighbor at *NW* and that is adjacent to a tile of a row r of t's column; if no such tile existed, the structure would be a block already. We distinguish between the cases that (1) s is a southern neighbor of r, and that (2) s lies *NE* to a tile of r. In Case (1), the test series fails at the latest when the robot traverses row r by Condition 2. If in Case (2) the robot places a tile *NW* of s (by moving a tile in the row above r), the test series will fail by Condition 3. Otherwise, the *NW* neighbor of s will still be empty when r is traversed by the robot, in which case the test series fails by Condition 2. □

Note that since tiles are exclusively moved *SE*, the resulting block has at most D rows consisting of at most D tiles each, and therefore diameter $\mathcal{O}(D)$. Similar to the construction of a line, it can be easily seen that the runtime to construct a block is asymptotically optimal: Consider a line of tiles from *SW* to *NE*. In order to transform the initial structure into a block, a constant fraction of tiles needs to be moved by a distance linear in D.

7.2.3. Forming a Tree

So far we have been mainly focusing on how to *quickly* construct suitable intermediate structures. However, regarding potential practical applications, it may also be desirable to minimize the required work space. Whereas the previous structures are in many cases built almost completely outside of the initial configuration's convex hull, in this section we present an algorithm that builds a simply connected structure by exclusively moving tiles inside the structure's convex hull.

First we introduce some additional notation. An *overhang* is a set of vertically adjacent empty nodes such that (1) the northernmost node has a tile at *N*, (2) the southernmost node has a tile at *S*, and (3) all nodes have adjacent tiles at *NW* and *SW*. A *tree* is a connected tile configuration without an overhang. Examples of an overhang and a tree can be found in Figures 7.6a and 7.6d, respectively. Since the westernmost nodes of a hole are part of an overhang, a tree is simply connected. We define the *branches* of a column as the column's western adjacent columns, where two columns are called adjacent if at least two of their tiles are adjacent. Finally, a *local tree* is a column whose connected component, obtained by removing all of its eastern neighbors, is a tree.

In this section, we present an algorithm that transforms any initial tile configuration into a tree in $\mathcal{O}(n^2)$ rounds and without ever placing a tile outside the initial structure's convex hull. The pseudocode of the algorithm is given in Algorithm 3. From a high level, the algorithm works as follows. The robot first traverses the tile structure in a recursive fashion until it encounters an overhang. It then fills the overhang with tiles and afterwards restarts the algorithm. Once the whole structure

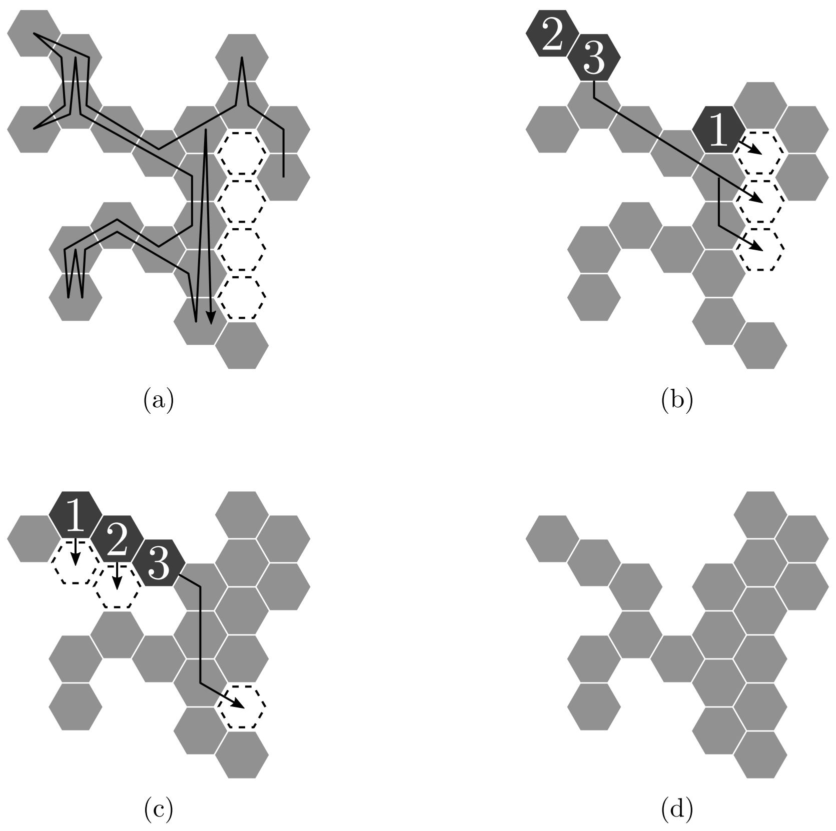


Figure 7.6.: (a) The traversal of the robot in an arbitrary connected structure, starting from an easternmost column, until detecting an overhang (dashed outlines). (b) The first three tiles are placed into the overhang. (c) Tiles (1) and (2) are moved south before (3) is brought into the overhang. (d) The resulting tree.

can be traversed without encountering any overhang, the tiles are arranged as a tree and the robot terminates.

Finding an Overhang More precisely, the robot does the following. In the `initialize` phase, it first successively moves to eastern columns until it reaches a locally easternmost column, and then moves to the northernmost tile of that column. Then the robot starts moving west with the `search_next_branch` phase. Starting at the easternmost column, the robot always searches the northernmost branch of each column by traversing the column from north to south. When it finds a branch, it moves into it and moves N until it reaches the northernmost tile of the branch, from where the process is continued. Eventually, the robot reaches a locally westernmost column that does not have any branches, and turns to the `check_overhangs` phase. The purpose of this phase is to check whether the current column has an adjacent eastern overhang by traversing the column from N to S. If so, the robot fills the overhang as described in the next paragraph and restarts the algorithm afterwards. Otherwise, the robot searches for an adjacent eastern column, of which there can be at most one, in the `move_E` phase. If there is none, the algorithm terminates. Other-

Algorithm 3 `Tree-Construction`

```
 1: phase initialize:
 2:    move to locally easternmost column
 3:    move N as far as possible
 4:    goto search_next_branch
 5: phase search_next_branch:
       branches farther north are local trees
 6:    move S until
 7:       case NW or SW occupied then
 8:          move NW (or SW)
 9:          move N as far as possible
10:       case reached column's end then
11:          goto check_overhangs
12: phase check_overhangs:
       current column is a local tree
13:    move N until end
14:    move S until
15:       case found eastern overhang then
16:          goto get_tile_N
17:       case reached column's end then
18:          goto move_E
19: phase move_E:
       column is local tree without overhang
20:    move N until
21:       case SE or NE occupied then
22:          move SE (or NE)
23:          if S occupied then
24:             move S
25:             goto search_next_branch
26:          else
27:             goto check_overhangs
28:       case reached column's end then
29:          terminate
30: phase get_tile_N:
31:    move N as far as possible
32:    if NW empty or SW occupied then
33:       goto bring_tile
34:    else
35:       goto get_tile_NW
36: phase get_tile_NW:
37:    if NW occ., but SW & N empty then
38:       move NW
39:    else if N occupied then
40:       goto get_tile_N
41:    else if SW occupied then
          there cannot be a tile at S
42:       lift tile and move it S
43:       if S or SE occupied then
44:          move NE
45:          goto bring_tile
46:       else
47:          move NE
48:    else
          locally northwesternmost tile
49:       goto bring_tile
50: phase bring_tile:
51:    lift tile and move S, SE (in that
       precedence) until there is tile at NE
52:    move S until there is no tile at SE
53:    move SE and place tile
54:    if S occupied then
       overhang was filled
55:       goto initialize
56:    else
57:       move SW
58:       goto get_tile_N
```

wise, the robot enters the eastern column at the southernmost tile that is adjacent to the robot's current column. If the entered tile has a southern neighbor, which is consequently not adjacent to the robot's previous column, there might be branches to traverse further south, and the robot continues with the `search_next_branch` phase. Otherwise, the entered column must be a local tree, and the robot can continue with the `check_overhangs` phase. An illustration of such a traversal can be found in Figure 7.6a.

Filling an Overhang Once the robot has found an eastern overhang, its goal is to repeatedly find tiles to lift and place into the overhang until it is filled. To do so, the robot first enters the `get_tile_N` phase; it will then alternate between this and the `get_tile_NW` phase, moving in a way that assures the robot to find its way back into the overhang. More specifically, in the `get_tile_N` phase the robot moves N as long as there is a tile at N, and in the `get_tile_NW` phase it moves NW as long as there is a tile at NW and no tile at SW or N. The robot's path either ends (1) in the `get_tile_N` phase at a tile that does *not* have a NW neighbor or *does* have a SW neighbor, in which case the tile is picked to bring into the overhang (e.g., Tile 1 in Figure 7.6b), (2) in the `get_tile_NW` phase at a locally northwesternmost tile, which would also be picked (e.g., Tile 2 in Figure 7.6b), or (3) in the `get_tile_NW` phase at a tile t that has a tile at SW (e.g., Tile 1 in Figure 7.6c).

In the third case, the node southern to t must be empty, and t is moved onto that node. If thereafter t has a neighbor at S or SE, the robot lifts t's NE neighbor t'. Otherwise, it moves onto t' and continues its search in the `get_tile_NW` phase. Both situations are illustrated in Figure 7.6c: First, the tile labeled 1 is moved south. As this tile does not have a neighbor at S or SE, the robot continues at the tile labeled 2, which, after having been moved south as well, has a southern neighbor. Finally, the robot picks the tile labeled 3.

After having picked a tile to bring into the overhang, the robot returns to its originating column c by moving S and SE in this order of precedence in the `bring_tile` phase. As the robot has never stepped on a tile with a southern neighbor outside of c, and has never performed a SW movement, it thereby precisely retraces its search path, and the first tile it encounters that has a NE neighbor must lie in c. The robot continues to bring tiles as described until the overhang is filled, in which case it again turns to the `initialize` phase.

Lemma 7.9 (Finding an Overhang)**.** *If the algorithm is executed by the robot starting on a tree, the robot traverses the tree completely and terminates within $\mathcal{O}(n)$ rounds. Otherwise, the robot finds an adjacent eastern overhang of a local tree within $\mathcal{O}(n)$ rounds.*

Proof. We first show the first part of the lemma. Assume a configuration is a tree, in which case every column has at most one adjacent eastern column. Furthermore, there is exactly one column that does not have an eastern neighbor. Following the `initialize` phase, the robot first moves to the northernmost tile of that column and then turns to the `search_next_branch` phase. We show that the robot traverses the local tree of each column completely in a recursive fashion. Since the local tree of the easternmost column is the whole tree, this implies the claim.

We claim that upon entering a column c for the first time, (1) the robot first moves to the northernmost tile of c, (2) completely traverses the column's branches from north to south, (3) verifies that c does not have any overhang, and then (4) enters c's adjacent eastern column c' via the southernmost tile of c' that is adjacent to c. If there is no such tile, then the robot has traversed the whole tree and terminates in Line 29.

First, note that (1) holds due to Line 3 if c is the initial column, or Line 9 if c is reached via an eastern column c'. We prove the other claims by induction on the depth of the local tree of c. For the base case of the induction, assume that c does not have any branches. Then the robot traverses c from north to south (Line 6) and immediately turns to the `check_overhangs` phase (Line 11). It then traverses the column once more to verify that is has no overhangs. Afterwards, by following the `move_east` phase, the robot traverses c from south to north until it reaches the southernmost tile of c' adjacent to c.

Now assume c has branches. When first entering c, the robot moves to the column's northernmost tile, and then enters the northernmost branch of c following the `search_next_branch` phase. By the induction hypothesis, it eventually reaches c again via the southernmost tile t of c that is adjacent to that branch in the `move_E` phase. If there are branches further south, t must have a southern neighbor and the robot continues to search for the next branch (Line 23). Following the above procedure, the robot traverses all branches of c until it eventually reaches the southernmost tile of c and turns to the `check_overhangs` phase through Lines 11 or 27. As there are no overhangs, it enters its adjacent eastern branch c' via the southernmost tile of c' that is adjacent to c. We conclude that if the configuration is a tree, then the robot moves through the whole structure and eventually terminates in the easternmost column.

If otherwise the configuration is not a tree, the robot will traverse the structure as described above until it eventually detects an eastern overhang in some column c during the `check_overhangs` phase. Since the robot must have traversed all branches of c, at this point c is a local tree.

It is easy to see that in any case each tile is visited no more than 6 times. Therefore, the robot halts within $\mathcal{O}(n)$ rounds. □

Lemma 7.10 (Filling an Overhang). *After detecting the northernmost eastern overhang of a column c in the* `check_overhangs` *phase, the robot will fill it and then turn to the* `initialize` *phase. At all times, the structure remains connected and c remains a local tree.*

Proof. First, after encountering the overhang, the robot turns to the `get_tile_N` phase in Line 16 and moves to the northernmost tile r of c. If r does not have a neighbor at *NW* or does have a neighbor at *SW*, then the robot lifts r and, by following the `bring_tile` phase, places it at the northernmost node of the overhang. As c has an overhang, and thus consists of at least two tiles, the robot can only disconnect the tile structure by removing r if there is a tile *NE* but not *SE* of r's previous position, in which case r is directly placed *SE*, reconnecting both parts (Tile 1 in Figure 7.6b).

If otherwise r has a neighbor at *NW* but not at *SW*, the robot initiates a search for a safely removable tile by turning to the `get_tile_NW` phase in Line 35. First, the robot moves *NW* as long as there is a tile at *NW*, no tile at *SW*, and no tile at *N*. If it reaches a tile that has a neighbor at *N* (Line 39), it again turns to the `get_tile_N` phase and continues as above. Since the robot only moves *N* and *NW*, and the tile set is finite, it eventually faces one of three situations. We show that in all three situations the robot identifies a safely removable tile.

In the first situation, the robot reaches a locally northwesternmost tile during the `get_tile_NW` phase (Line 48, Tile 2 in Figure 7.6b after Tile 1 has been moved). Since this tile must have been reached via its *SE* neighbor, it can be safely removed.

In the second situation, the robot encounters a northernmost tile t that has no tile at *NW* or a tile at *SW* during the `get_tile_N` phase (Line 32, Tile 3 in Figure 7.6b after 1 and 2 have been moved). As t lies in a branch of c, in which there cannot be any overhangs by the discussion of Lemma 7.9, t cannot have a neighbor at *NE* and thus can be safely removed.

In the third situation, the robot reaches a tile t that has no neighbor at *N*, but at SW, during the `get_tile_NW` phase (Line 41, Tile 1 in Figure 7.6c). t must have been reached via its *SE* neighbor t', which, consequently, cannot have a neighbor at *N* nor SW. First, the robot moves t one step south. If t does not have a neighbor at *S* nor *SE*, then the robot moves onto t' and continues with the `get_tile_NW` phase (Tile 1 in Figure 7.6c after having been moved south). Note that the same situation might happen again for t', and may even repeat for every tile of the corresponding row, which will be moved south one after the other (Tiles 1 and 2 in Figure 7.6c).

However, the robot must eventually move a tile t south that faces an adjacent tile at *S* or *SE*. In this case, its adjacent tile t' at *NE* is lifted (Line 45). We have to show that the structure remains connected and c remains a local tree. We distinguish two cases. First, if t only has a neighbor at *S*, then $t' \neq r$ (Tile 2 in Figure 7.6c after 1 and 2 have been moved south). As there are no overhangs in the local tree of c and since the robot never moves SW, t' cannot have a neighbor at *NE*. Furthermore, the tile south of t must be a tile of a different branch than the branch of r. Therefore, connectivity of the structure is preserved after lifting t'. The new connection established by moving t can also not introduce any overhangs into c's local tree. Second, if t has an adjacent tile at *SE* after having been moved, then its *NE* neighbor t' is either r, or it is a different tile that does not have adjacent tiles at *NW*, *N*, and, since c does not have overhangs, *NE*. In either case, t' can be safely removed and brought into the overhang.

We finally argue that the tile that the robot picks is correctly placed into the overhang. If the robot picks a tile in the first or second situation above, then no tile traversed by the robot can have a tile at *NE* (except for r), and the robot has never moved SW. If the robot picks a tile in the third situation, then the same holds for the search path up to the position of the picked tile. Therefore, moving *S* and *SE* (in that precedence) in the `bring_tile` phase (Line 51) precisely retraces the robot's search path, and brings the robot back into column c. As c has an eastern overhang, the robot will encounter a tile that with an adjacent tile at *NE* in c throughout this traversal, and, since it cannot encounter such a tile before, can safely assume that it reached column c. To find the overhang, it now simply needs to move south until it sees an empty node v at *SE*, and can then place its tile at v. If v has an adjacent tile at *S* (Line 54), the overhang has been filled and the `initialize` phase is entered. Otherwise, the robot continues until all nodes of the overhang are filled. □

Lemma 7.11 (Convex Hull)**.** *Following the algorithm, the robot never places a tile outside of the convex hull of the initial configuration.*

Proof. The robot does not leave the convex hull by placing a tile t into an overhang. The only other movement of t occurs in Line 42 in the situation depicted in Figure 7.6c. Here, the robot has reached t by moving a sequence of *NW* and *N* steps starting at the northernmost tile of a column c with an adjacent eastern overhang. Since c has at least 2 tiles, the robot will not move t outside the configuration's convex hull. □

Using the previous lemmas, we are now ready to prove the following theorem.

Theorem 7.12 (Tree Formation)**.** *A single robot can transform any connected tile configuration into a tree in $\mathcal{O}(n^2)$ rounds and without placing a tile outside the initial configuration's convex hull.*

Proof. If the configuration is a tree, then by Lemma 7.9 the robot terminates within $\mathcal{O}(n)$ rounds. Therefore, assume the configuration is not a tree. Lemma 7.9 states that the robot will find an overhang, which, by Lemma 7.10, will subsequently be filled. Afterwards, the algorithm will be restarted (Line 55). We define a *possible overhang* as a maximal but finite column of vertically adjacent unoccupied nodes in the initial tile configuration. Nodes that are not part of an overhang from the beginning can only ever become part of one if they are inside a possible overhang. Since the robot does not create any new possible overhangs, it can only fill finitely many overhangs before the tile configuration is arranged as a tree. After performing a last traversal through the tree, the robot terminates.

We now turn to the runtime of the algorithm. Clearly, there are only $\mathcal{O}(n)$ possible overhangs in an initial configuration. Since traversing the structure before finding an overhang takes $\mathcal{O}(n)$ rounds by Lemma 7.9, and the algorithm is restarted at most $\mathcal{O}(n)$ times, the total number of rounds needed for traversing the structure is $\mathcal{O}(n^2)$. Since tiles are only moved *S* or *SE* and, by Lemma 7.11, tiles are never moved outside the initial configuration's convex hull, each tile is moved at most $\mathcal{O}(n)$ steps. Furthermore, for every search step in direction *N* and *NW* in the `get_tile_N` phase and `get_tile_NW` phase, the robot either moves a tile by one step in the opposite direction, or moves a tile *S* and takes a single step *NE*. Therefore, $\mathcal{O}(n^2)$ rounds are needed for searching and moving tiles, which implies that the robot terminates within $\mathcal{O}(n^2)$ rounds. □

7.3. Forming a Triangle

We will now describe how the robot can transform an intermediate structure, more precisely a block, into a *triangle*. A triangle consists of columns whose northernmost tiles form a row, and each column consists of exactly one tile more than its eastern adjacent column. Depending on whether n is a triangular number, the westernmost column of the triangle may only partially be filled (see right picture in Figure 7.7). In the following, we assume that a block has already been built. It can be easily seen that a line and a tree can be transformed in a similar way. The triangle is built by repeatedly taking the easternmost tile of the block's northernmost row, carrying it south to the *vertex* of the forming triangle (i.e., the easternmost column consisting of one tile), and adding it to the westernmost column of the triangle (see Figure 7.7).

First, the robot creates the vertex of the triangle by placing a tile on the node v below the westernmost column of the block. A second tile is then placed *NW* of v. Every other tile of the triangle is then placed as follows. The robot brings a tile to the triangle's vertex, and then walks *NW* and *S* (in that precedence) until there is no tile in any of these directions. If there is a tile at *SE*, the robot moves one step *S* and places the tile. Otherwise, the robot moves *N* to the top of the column, takes one step *NW*, and places the tile. In this manner, the robot continues to

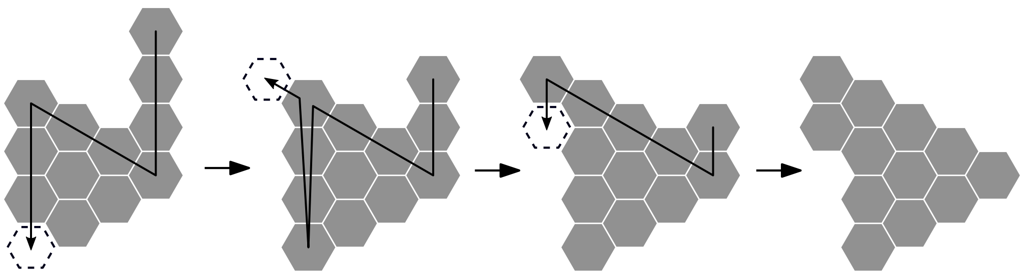


Figure 7.7.: Snapshots of triangle formation. If the number of tiles is not triangular, the final column will not be completely filled.

extend the triangle tile by tile until the block reduces to the triangle's vertex. From Theorem 7.8, and since each tile can be brought and placed within $\mathcal{O}(D)$ rounds, we conclude the following theorem.

Theorem 7.13 (Triangle Formation)**.** *A single robot can transform any connected tile configuration with diameter D into a triangle and terminate after $\mathcal{O}(nD)$ rounds.*

In case that the initial configuration's diameter is $D = \mathcal{O}(n^{1/2})$, a triangle can be constructed in $\mathcal{O}(n^{3/2})$ rounds. It is not hard to see, using similar arguments as in the previous sections, that the runtime is asymptotically optimal.

7.4. Towards Multiple Robots

As a first step towards extending our algorithms to the multi-robot case, we show that multiple robots can cooperatively construct a triangle using a line as an intermediate structure. We believe that some of our ideas may also be useful to solve more difficult problems. First, we present and discuss the underlying model assumptions. Then, we briefly describe how the line formation algorithm can be adapted for multiple robots. We experimentally show that the construction of a line can be sped up significantly by using multiple robots. Finally, we describe a simple algorithm to transform a line into a triangle using multiple robots.

Model Discussion We consider the following extension of our model to incorporate multiple robots. For brevity, we leave out a formal definition. Each node is occupied by at most one robot at any time. We adapt our notion of connectivity and require all robots to be adjacent to occupied nodes, and the subgraph of G induced by all occupied nodes T and the positions P_c of all robots carrying tiles to be connected. Therefore, we may have lines of robots attached to the side of the structure only if these robots carry tiles. In the `Look` phase, for each adjacent node a robot can observe whether the node is occupied by another robot, and determine the state of that robot. It then uses this information to determine its next state and move in the `Compute` phase, and may change the state of each adjacent robot. Finally, in the `Move` phase, the robot may either perform one of the five actions described in the model section, or pass a carried tile to an adjacent robot that does not yet carry a tile.

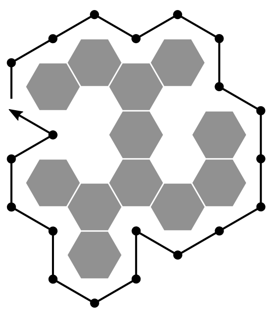


Figure 7.8.: A clockwise boundary traversal that does not pass through bottlenecks.

We assume an asynchronous model in which robots are *activated* in an arbitrary sequence of *activations*, where a robot performs exactly one look-compute-move cycle before the next robot is activated (see, e.g., [Day+19]). A round is over when each robot has been activated at least once. For simplicity, we not only assume that all robots have the same chirality, but also share a common compass. In fact, lifting this restriction imposes difficult challenges outside the scope of this thesis, since symmetry breaking is very hard in our deterministic model. We leave this issue as a future research question.

Distributed Line Formation In order to extend the line algorithm to work with multiple robots, we propose three main modifications to the line formation algorithm. The pseudocode of the algorithm can be found in Algorithm 4. First, a robot r that carries a tile and is blocked in *S* or *SE* direction by a robot that searches for a tile can pass its tile and state to the blocking robot. If afterwards r stands on a tile, it turns to the search phase. Otherwise, r has left the tile structure (we say it is *hanging*) and subsequently traverses the boundary of the structure in clockwise order, maintaining its connectivity to the outline of the tile structure until it reaches an empty tile to step on. We make sure that no hanging robot is disconnected from the tile structure by a robot picking up a tile by performing additional checks.

Second, we ensure that no hanging robot ever ends up in a deadlock whilst traversing the boundary by avoiding to walk into *bottlenecks*, i.e., empty nodes with tiles on two non-adjacent sides. A traversal that avoids bottlenecks is depicted in Figure 7.8. Finally, in order to eventually let each robot detect that the line has been built, we slightly modify the way tiles are moved. More specifically, we do not immediately move a lifted tile *SE* and place it at the first empty position in the column as in the single-robot algorithm. Instead, after lifting a tile, a robot first walks *S* until it actually encounters a column to its east, and only then moves *SE* to place its tile in that column. If there is no such column, the tile is simply placed at the bottom of the current column. A robot that has placed its tile without encountering a column to the east or west terminates. Note that a terminated robot r may block a robot r' coming from *N* from placing its tile at the bottom of the line. If we want to place all remaining carried tiles, we can, for example, simply pass the tiles down through the finished robots and let the southernmost of these robot place the tiles at the bottom of the line; in this case, final termination has to be detected slightly differently.

Simulation Data Although correctness can be proven for this multi-robot approach, it is difficult to make any runtime guarantees. This is due to the fact that,

Algorithm 4 `Distributed-Line-Formation`

Remark: Whenever the movement of a robot r that carries a tile is blocked by a robot r' that does not carry a tile, r passes its tile and state to r' and goes to phase `search`*, if r stands on a tile, and to phase* `hanging`*, otherwise. In the latter case, r sets* `next_dir` *to* N*. If r is blocked by a robot that also carries a tile, r waits.*

```
phase search:
    find a locally northwesternmost tile by moving $NW$, $SW$, and $N$
    wait whenever the next tile is already occupied by a robot
    if removing the tile does not locally disconnect a hanging robot then
        lift tile
        is_line ← TRUE
        goto carry_orig_col
phase carry_orig_col:                     ▷ move tile S until an eastern column is reached
    if there is western or eastern (carried) tile then                    ▷ not a line yet
        is_line ← FALSE
    if (carried) tile at $NE$ or $SE$ then                         ▷ reached eastern column
        move $SE$ and goto carry_next_col
    else if (carried) tile at $N$ and not standing on tile then           ▷ end of column
        place tile
        if is_line then terminate else goto search
    else if terminated robot at $S$ then                             ▷ line has been built
        terminate
    else
        move $S$
phase carry_next_col:                       ▷ move S as far as possible and place tile
    if standing on tile then
        move $S$
    else
        place tile and goto search
phase hanging:                     ▷ traverse the boundary until an empty node is reached
    if adjacent to empty tile then
        move there and goto search
    for $i$ in $(-2, -1, 0, 1, 2)$ do   ▷ choose next direction according to clockwise traversal
        let $d$ ($d'$) be the direction after turning $i$ ($i+1$) steps in clockwise order
        starting at next_dir
        if no tile at $d$ and tile at $d'$ then
            next_dir ← $d$
            if no robot at next_dir then
                move next_dir
            break
```

when there are many robots compared to the number of tiles, robots will often be blocked by others and must wait to make progress. However, as a first step, we experimentally evaluated the number of rounds until all robots halt. The results for $n = 10000$ and a varying number k of robots can be found in Figure 7.9. We conducted 50 simulations for each k, each initialized with a randomly generated tile configuration on which the robots were randomly placed. The robots were activated in a random order, each exactly once in every round.

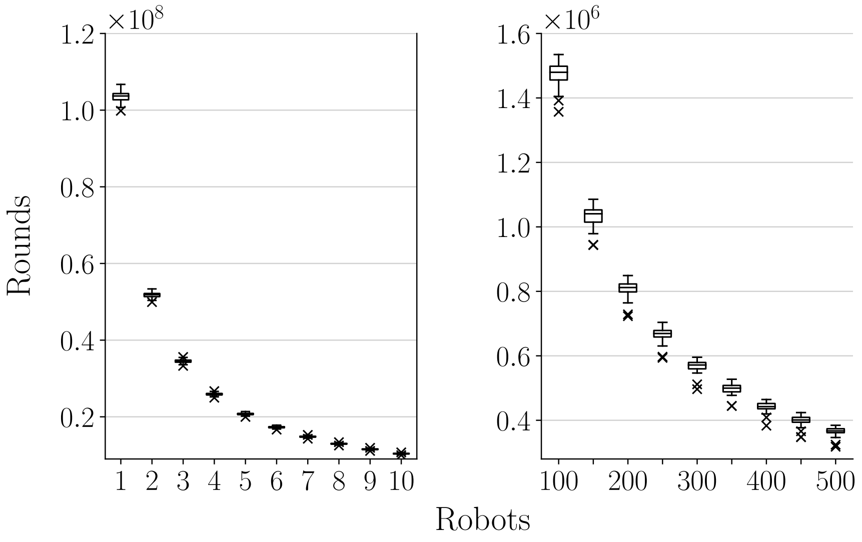


Figure 7.9.: Simulation results for 10000 tiles.

Each tile configuration was generated by the following procedure: First, randomly choose $10000 \cdot 16^2/2.02$ nodes of an equilateral parallelogram with side length $\sqrt{10000} \cdot 16$ to be occupied by a tile. Then, repeat the experiment until the largest connected component of the generated tile set contains at most 10500 tiles. The final configuration is obtained by repeatedly removing random tiles from the component whose removal does not disconnect the structure until 10000 tiles remain.

The simulations show that using a reasonably small number of robots significantly reduces the required number of rounds compared to using a single robot. The curve on the left part of Figure 7.9 first decreases rapidly (e.g., going from one to two robots essentially halves the runtime). However, for a large number of robots the benefit gained from employing more robots is almost negligible (right part of Figure 7.9). This phenomenon can likely be explained by the fact that the likeliness of robots waiting on each other increases with the number of robots. Nevertheless, these preliminary results suggest that the model indeed allows multi-robot algorithms whose runtime drastically decreases if the number of robots is reasonably small.

Distributed Triangle Formation If the structure is arranged as a line, a triangle can easily be built in a distributed manner: In order to retrieve tiles, the robots traverse the line from south to north. Once a robot reaches the northernmost tile, it waits until there is no robot north of it anymore, then lifts the tile and carries it to the triangle by following the boundary of the structure in clockwise order. The next position to place a tile into the triangle can easily be found, and the robot returns to the line by moving to the triangle's vertex. Whenever a robot's move is hindered by another robot, it simply waits.

Arguably, this algorithm makes use of multiple robots more effectively than the distributed line algorithm: No robot is ever forced to leave the tile structure, or to wait in order to preserve connectivity. This higher degree of coordination between the robots is facilitated using additional knowledge of the tile structure. However, coming up with a similar strategy for arbitrary structures seems to be rather difficult.

7.5. Outlook

This chapter shows that a *single* robot can solve complex shape formation tasks despite its very limited capabilities. Clearly, the ultimate goal of hybrid programmable matter is to leverage multiple robots. Whereas our experimental study indicates that shape formation can, in principle, benefit from the power of multiple robots, thorough theoretical work is yet to be done. Specifically, we believe that for the shape formation tasks described in this chapter a *linear* speedup in the number of robots should be feasible by using a much more coordinated approach. If there are too many robots compared to the number of tiles, then such a speedup will probably not be possible; however, as long as $k = \mathcal{O}(\sqrt{n})$, the utility of each additional robot should not be diminished by other robots blocking it.

Future work may also involve the formation of more complex shapes. Specifically, the *universal shape formation* approach, which has been studied for the Amoebot model [Der+16; Di +20], should be viable in our model as well. Further algorithms for hybrid programmable may also exploit the capability of the robots to simulate binary counters [Day+20] or even Turing machines [Fek+21] using either a collection of finite automata or by modifying the tile structure. As already indicated in Section 7.1, another way to increase a robot's power is to equip it with *pebbles*. This possibility is further explored in the following chapter, where we consider *shape recognition problems* with pebbles.

8 Shape Recognition in Hybrid Programmable Matter

Whereas our work on hybrid programmable matter assumes a fully reliable system in which no errors occur, it is unreasonable to expect that this will be the case in practice. The before-mentioned DNA tile-based approach to programmable matter [Pat14] for example, which may at some day be helpful to realize hybrid programmable matter, is known to be particularly error-prone. Numerous techniques have been studied to reduce the error rates (e.g., [EW13]), but the research typically focuses on designing DNA tiles or assembly processes that reduce the error of incorrect attachments. In our hybrid approach, it may be possible instead to use robots to make the system more reliable. For the shape formation problem studied in the previous chapter, for example, the robots may be able to cope with tiles being torn away from the structure by continuously repairing the shape under construction. To do so, however, they must be able to *detect* places that need repair.

In this chapter, we generalize such problems as *shape recognition* problems and investigate their complexity. As before, we focus on a *single* robot, but may give the robot additional pebbles to mark positions on the tile structure. Specifically, we consider the problem of detecting whether the tile structure is a parallelogram with some given side ratio, and investigate how many pebbles are necessary and sufficient to correctly determine certain side ratios.

Underlying Publication The chapter is based on the following publication.

> R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, and C. Scheideler. “Shape Recognition by a Finite Automaton Robot”. In: *Proceedings of the 43rd International Symposium on Mathematical Foundations of Computer Science (MFCS).* 2018, 52:1–52:15 [Gmy+18a]

A preliminary version of our work has been presented at the EuroCG 2018 Workshop [Gmy+18b].

Contribution and Outline All of our results are achieved in the model described in Chapter 7 with a single robot and a specific number of pebbles. In Section 8.1, we begin with testing whether a given tile formation is of a certain simple shape without any pebble, more precisely, a line, a triangle, a hexagon, or a parallelogram. Then, we turn to the much more difficult problem of deciding for a given function $f(\cdot)$ whether the longer side of a parallelogram has length $f(h)$, where h is the shorter side's length. An overview of our results is given in Table 8.1, in which we state functions $f(\cdot)$ the robot is able or not able to decide given a certain number

Pebbles	Possible	Impossible	Section
0	$f(x) = ax + b$	$f(x) = \omega(x)$	8.2.1
1	$f(x) = a_d x^d + \ldots + a_0$	$f(x) = \omega(x^{6s+2})$	8.2.2
2	$f(x) = \underbrace{c^{c^{\cdot^{\cdot^{\cdot^{c^x}}}}}}_{\alpha x+\beta}$	—	8.2.3
k	$f_k(x)$	$f_{k+1}(x)$	8.2.4

Table 8.1.: This table summarizes the results for recognizing whether a given parallelogram has height h and length $\ell = f(h)$ given a certain number of pebbles. The variables $a \in \mathbb{Q}_{\geq 1}$, $b \in \mathbb{Z}$, $c, d, \alpha, \beta \in \mathbb{N}_0$, and $a_i \in \mathbb{Z}$ for all i are constant. $s = |Q|$ is the number of the robot's states.

of pebbles. Our ultimate goal is to investigate the computational capabilities of a simple robot concerning shape recognition, and to what extent the robot can benefit from employing pebbles.

As we do not make any assumptions on the length of the shorter side h of the parallelogram, we cannot assume that a robot with only a constant number of states can count up to h, even less so evaluate the function $f(h)$. Thus, if the robot does not have pebbles at its disposal, its only option is to make use of the environment's geometry. For example, the robot can "measure" h tiles along the longer side of the parallelogram by starting in a corner and moving diagonally until reaching the opposite boundary. Furthermore, the robot can measure $2h$, $3h$, or even ah tiles, for any constant a, along the longer side.

In Section 8.2.1, we develop this intuition further and show that the robot can decide whether the longest side of the parallelogram is $\ell = ah + b$, where a and b are constants. On the other hand, we show that the robot without pebbles is not able to recognize any superlinear function. In Sections 8.2.2 and 8.2.3, we show that we can tremendously increase the robot's computational power by giving it a single pebble or two pebbles, respectively. More precisely, having the ability to mark any tile with a single pebble allows the robot to recognize any polynomial function of constant degree. Being equipped with two pebbles gives the robot the ability to recognize power tower functions, where the height of the power tower is constant or even linear in h. Finally, in Section 8.2.4 we show that for any number k of pebbles there exists a function that requires k pebbles to be decided by the robot.

Additional Related Work Whereas shape formation has been extensively studied in various models, to the best of our knowledge, the closely related problem of shape *recognition* has never been explicitly considered in our setting. However, solving problems by traversing a tile structure with simple agents has been investigated in many different areas. For an overview of related work in the two areas of shape formation and agents on graphs see Chapter 7. Notably, [Sha74] considers the problem of deciding whether a structure is simply connected.

For many problems for agents on graphs, it has also been investigated whether pebbles can be helpful. This question is particularly well-studied for the classical

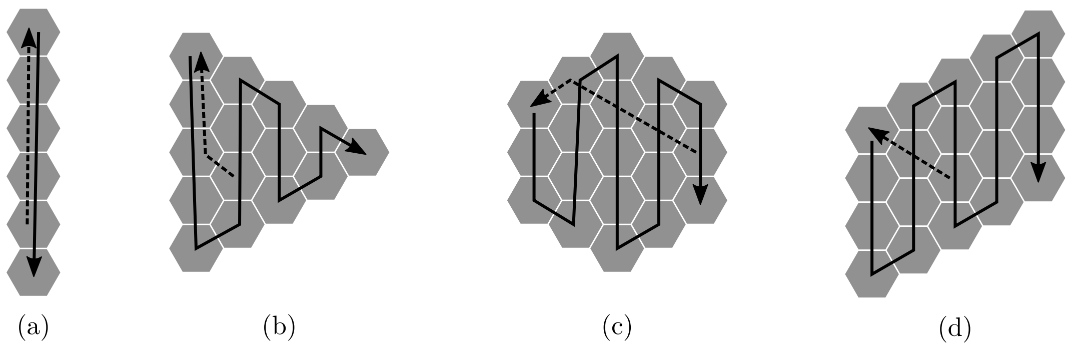


Figure 8.1.: The traversal of the robot to detect a line, a triangle, a hexagon, and a parallelogram. The dashed line is the path of the robot until it reaches a locally northwesternmost tile, and the solid line is the subsequent traversal of the complete structure.

Network Exploration Problem (see, e.g., [Das13]). For example, it is known that a robot cannot explore all planar graphs [Fra+05] unless it can store $\Omega(\log n)$ bits. A finite automaton can also not find its way out of any planar labyrinth [Bud78], even if given a single pebble [Hof81]. However, a robot with two pebbles can solve the problem [BK78].

8.1. Recognizing Simple Shapes

First, we show that a single robot can easily detect whether the initial structure is a line, a triangle, a hexagon, or a parallelogram. A depiction of the traversal of the robot in these structures can be found in Figure 8.1.

Line To test if a given tile shape is a line, the robot first moves to a locally northwesternmost tile by moving *NW*, *N*, and *SW* until no longer possible. From there, it can determine the (potential) orientation of the line, i.e., whether it is oriented in direction *S*, *SW*, or *SE*. If, for example, the line is oriented in direction *S* as shown in Figure 8.1a, the robot simply traverses the structure in that direction until no longer possible. If it ever encounters a tile to the west or east of any traversed tile, the structure is not a line.

Triangle, Hexagon, Parallelogram All other structures are traversed in a “snake-like fashion”. For a triangle, the robot first moves to a locally northwesternmost tile. After having arrived there, it can determine the orientation of the triangle, i.e., whether it points westwards or eastwards. W.l.o.g, assume that the triangle points eastwards as in the example in Figure 8.1b. The robot then traverses the shape column by column from west to east (recall that a column is a maximal sequence of tiles from *N* to *S*). More precisely, it first moves *S* as far as possible, then takes one step *NE*, moves *N* as far as possible, and finally takes an additional step *NE*. The procedure is repeated until a *NE* movement is no longer possible. By performing local checks when moving through a column and whenever a new column is entered, the robot can verify whether the tile shape is a triangle. As depicted in Figures 8.1c and 8.1d, a hexagon and a parallelogram are traversed in the same way, only differing in the specific checks that need to be performed.

Observation 8.1 (Simple Shape Recognition)**.** *A robot without any pebble can detect whether the initial tile configuration is a line, a triangle, a hexagon, or a parallelogram.*

8.2. Recognizing Parallelograms with Specific Side Ratio

As noted in Observation 8.1, a single robot without pebbles can verify whether a given shape is a parallelogram. To investigate the computational power of a finite automaton, in this section we consider the problem of deciding whether a parallelogram has a given side ratio. Additionally, we examine how pebbles can be helpful to decide more complex side ratios.

We assume w.l.o.g. that the robot needs to detect whether the given tile configuration is a parallelogram that is axis-aligned along the north and north-east direction as shown in Figure 8.1d. Slightly changing our notation from the previous chapter, we define a *row* as a maximal sequence of tiles from SW to *NE*. Let h be the size of each column, i.e., the parallelogram's *height*, and ℓ be the size of each row, i.e., the parallelogram's *length*. W.l.o.g., we assume that $h \leq \ell$; otherwise we can simulate our algorithms on a rotated and reflected version of the parallelogram. We enumerate the columns of the parallelogram from 0 to $\ell - 1$ growing in direction *NE*.

8.2.1. A Robot without any Pebble

First, we point out that a single robot can detect whether the structure is a parallelogram in which its length ℓ is a linear function of its height h. Our algorithm is based on the idea that if the robot moves *SE* as far as possible starting at the northernmost tile of column 0, it will end up at the southernmost tile of column $h - 1$. Therefore, if ℓ is a multiple of h, then the precise factor can be derived from the number of times such a diagonal movement is possible.

Theorem 8.2 (Linear Functions)**.** *A single robot can detect whether the tile configuration is a parallelogram with $\ell = ah + b$ for any constants $a, b \in \mathbb{N}$.*

Proof. First, the robot verifies whether the structure is a parallelogram. If so, it moves to the northernmost tile of column 0. The tile structure is then traversed in two stages, where the first stage measures the distance ah and the second stage measures b. More precisely, in the first stage the robot moves in a “zig-zag” fashion as depicted in Figure 8.2 by performing the following movement in a loop: (1) move *SE* as far as possible, (2) move *N* as far as possible, and (3) make one step *NE*. After having performed the complete sequence of *SE* movements a times, the robot moves on to the second stage, in which it makes an additional b *NE* steps.

If the robot reaches the easternmost column before completing the above procedure, or finally halts on a tile with an adjacent tile at *NE*, it terminates with a negative result. Otherwise, the test is successful. It is easy to see that $\ell = ah + b$ if and only if the robot terminates with a positive test result. □

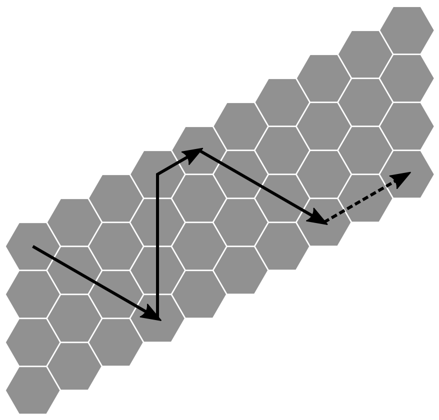


Figure 8.2.: A parallelogram with height 4 and length $10 = 2 \cdot 4 + 2$. The solid arrows indicate the movement of the robot in the first stage of the algorithm described in Theorem 8.2 to measure $a = 2$. The dashed arrow shows the final *NE* movement to measure $b = 2$.

We remark that the algorithm can be adapted for $b \in \mathbb{Z}$, i.e., b can also be a negative integer. To achieve that, we halt the execution of the first stage once the robot has performed $|b|$ *SE* steps, then move *SW* as far as possible, and continue the first stage from there. Then, $\ell = ah + b$ if and only if the robot halts on the southernmost tile of column $\ell - 1$.

Furthermore, the algorithm can be extended to apply in the case of a rational $a \in \mathbb{Q}_{\geq 1}$. Let $a = p/q$ be an irreducible fraction, where $p, q \in \mathbb{N}$ and $p \geq q$. Instead of moving in a zig-zag fashion in the first stage, the robot alternates between moving p steps *NE* and q steps *S*. The second stage is unchanged. To exactly end up at the southernmost tile of column $\ell - 1$ if and only if $\ell = ah + b$, the robot needs to skip the very first *NE* and *S* step of the first stage. By combining these two modifications, we get the following corollary.

Corollary 8.3. *A single robot can detect whether the tile configuration is a parallelogram with $\ell = ah + b$ for any constant $a \in \mathbb{Q}_{\geq 1}$ and constant $b \in \mathbb{Z}$.*

We have shown that a single robot can determine whether the length of a parallelogram is given by a certain linear function of its height. However, that is as much as one robot can hope for. Indeed, a single robot is not able to decide whether the length of the parallelogram is given by a superlinear function of its height, as the following theorem states.

Theorem 8.4 (Superlinear Functions)**.** *A single robot without any pebbles cannot decide whether the tile configuration is a parallelogram with $\ell = f(h)$ in case that $f(x) = \omega(x)$.*

Proof. Suppose there is an algorithm that lets the robot decide whether the tiles are arranged as a parallelogram with $\ell = f(h)$ for some superlinear function $f(x) = \omega(x)$. Let s be the total number of states used by the robot in the algorithm. Choose h large enough such that $\lfloor \frac{f(h)-2}{h+2} \rfloor > s$, which can be done since $f(h) = \omega(h)$.

Consider the execution of the algorithm on a parallelogram P with height h and length $\ell = f(h)$. We will show that there exists another parallelogram with height h and length greater than $f(h)$ on which the robot eventually terminates in the same state as on P, which contradicts the assumption that the algorithm is correct.

We first place the robot on the northernmost tile of column 0. First, observe that if the robot does not visit column $\ell - 1$ during the execution of the algorithm, it cannot correctly detect the length of the parallelogram. Thus, the robot visits this column at least once.

Consider the execution of the algorithm as a sequence $(p_1, p_2, \dots)$ of tuples of nodes and states $p_i = (u_i, q_i)$. Denote as $\pi_1, \dots, \pi_m$ the subsequences of the execution of the algorithm, where each subsequence starts whenever the robot leaves a node of column 0 or $\ell - 1$, and ends when it enters a node of one of those columns. More specifically, the first and last node of each π_i is a node adjacent to column 0 or $\ell - 1$ and the node immediately before and after each π_i is a node of 0 or $\ell - 1$. Note that in each π_i the robot exclusively moves in the columns between column 0 and $\ell - 1$.

First, consider a subsequence π_i at the beginning of which the robot leaves, and at the end of which, enters the same column 0 (or $\ell - 1$). Let $p_i = (u_i, q_i)$ be the node-state tuple right before the robot enters the subsequence π_i when executing the algorithm on the parallelogram P. W.l.o.g., let u_i be a node of column 0, and let r be the index of its row when enumerating the rows from 0 to $h + 1$ from north to south, where row 0 refers to the empty nodes above P, and row $h+1$ refers to the empty nodes below P. Then, the robot would execute exactly the same subsequence π_i on a parallelogram P' with a larger length than the one of P if it was placed on the tile at column 0 and row r in P' in state q_i.

Next, consider a subsequence π_j at the beginning of which the robot leaves, w.l.o.g., column 0, and at the end of which it enters $\ell-1$. Since the robot completely traverses the tile structure from column 0 to $\ell - 1$, there must be a row R_j in which the robot steps on no less than $\lfloor \frac{f(h)-2}{h+2} \rfloor > s$ nodes between column 1 and $\ell - 2$. Therefore, there will be two nodes u_j and v_j in row R_j in which the robot appears in the same state. We specifically choose v_j in R_j as the *first* node for which there exists a node u_j that has been visited before and in the same state in π_j. Let c_u, c_v be the column indices of u_j and v_j, respectively. Note that we must have that $c_v > c_u$, as otherwise the robot would repeat its movements between u_j and v_j until it eventually reaches column 0, which cannot happen since π_j ends at column $\ell-1$. By the same argument, the robot will repeat its movement between u_j and v_j until it reaches column $\ell - 1$.

We exploit this observation in the following way. Define $d_j = c_v - c_u > 0$, and let (u_j, q_j) be the node-state tuple of the robot immediately before it enters the subsequence π_j and (u'_j, q'_j) be the node-state tuple of the robot immediately after the subsequence π_j is finished. Now consider a parallelogram Q with height h and length $f(h) + cd_j$, where $c \in \mathbb{N}_0$. If the robot starts in state q_j on the node in column 0 and the row of s_j in the parallelogram Q, it will move entirely between the westernmost and easternmost column of Q until it reaches the node of column $f(h) + cd_j$ in the row of u'_j in Q in state q'_j.

Now consider the execution of the algorithm on a parallelogram P' with height h and length $f(h)+\prod_j d_j$ for each subsequence π_j where the robot completely traverses the parallelogram between column 0 and $\ell - 1$. By the above argument, the robot will enter and leave those columns on the same tile and in the same state as in the execution of the algorithm on P. Thus, when executing the algorithm on the parallelogram P', the robot will ultimately terminate in the same state as if it was on P. Therefore, the robot cannot decide whether the initial tile configuration is a parallelogram with $\ell = f(h)$ for any superlinear function $f(x) = \omega(x)$. □

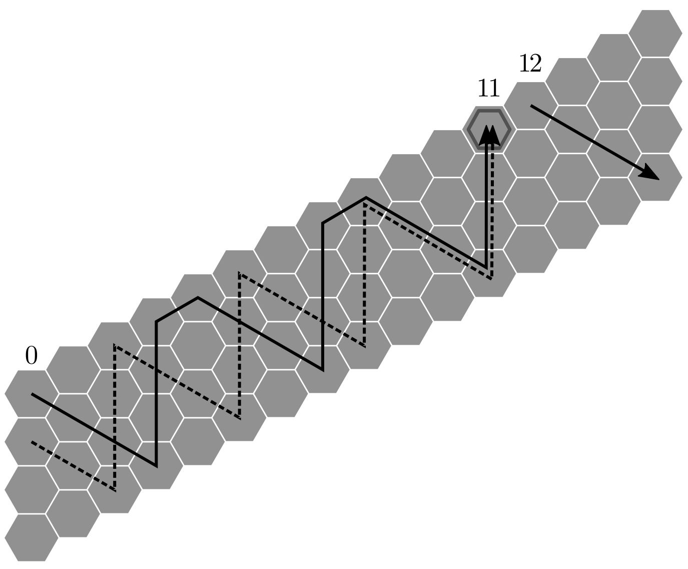


Figure 8.3.: A parallelogram with $h = 4$ and $\ell = 4^2 = 16$. The first column in which the normal zig-zag (solid line) and the truncated zig-zag (dashed line) both move north is column $11 = h(h-1)-1$. After the pebble has been placed there, the final diagonal movement shifts the robot eastwards by an additional h columns.

8.2.2. A Robot with a Single Pebble

In the following, we demonstrate that, in contrast to the negative result of Theorem 8.4, a single robot can decide any polynomial of constant degree. As an introductory example, consider $f(x) = x^2$. Our algorithm is based on a generalization of the observation that the *least common multiple* $\mathrm{lcm}(x, x-1)$ of x and $x-1$ is simply their product $x(x-1)$. Therefore, we can place a pebble at column $h(h-1)$ in the following way: First, we place a pebble at the northernmost tile of column 0. Then, we repeatedly move the pebble one step *NE* until it is reached with a *N* movement *both* during the zig-zag described in Theorem 8.2 as well as a "truncated zig-zag" that only moves the robot eastwards by $h-1$ instead of h steps (see Figure 8.3). This can only happen if for the current column index c of the pebble we have that $h \mid (c+1)$ (i.e., h divides $c+1$) and $(h-1) \mid (c+1)$. Since $\mathrm{lcm}(h, h-1) = h(h-1)$, this is true for the *first* time when $c+1 = h(h-1)$. Finally, since $h^2 = h(h-1)+h$, the robot can perform one additional diagonal move from the northernmost tile of column $h(h-1)$ and ends at the southernmost tile of column $\ell - 1$ if and only if $\ell = h^2$.

Theorem 8.5 (Polynomial Functions)**.** *A single robot with a pebble can decide whether the tile configuration is a parallelogram of height h and length $\ell = f(h)$ for any given polynomial $f(x) = a_d x^d + \ldots + a_0$, where $d \in \mathbb{N}_0$ and all $a_i \in \mathbb{Z}$ are constants.*

Proof. We define the *falling factorial* of x as $(x)_i := x(x-1)\cdots(x-i+1)$, and transform the input polynomial $f(x)$ into the form $f(x) = \alpha_d \cdot (x)_d + \alpha_{d-1} \cdot (x)_{d-1} +$

$\ldots + \alpha_0$. Note that since the $a_i \in \mathbb{Z}$ as well as $d \in \mathbb{N}_0$ are constants, the α_i are also constants from $\mathbb{Z}$. We will show that the robot can move the pebble in phases, by $|a_i \cdot (h)_i|$ steps in each phase i. Let $\mathrm{lcm}_i(x) := \mathrm{lcm}(x, \ldots, x - i + 1)$ and $g_i(x) := (x)_i/\mathrm{lcm}_i(x)$. From [HY08] it follows that $\mathrm{lcm}_i(x) \mid (x)_i$, and that $g_i(x)$ is periodic with period $\mathrm{lcm}(1, \ldots, i-1)$, i.e., $g_i(x) = g_i(x + \mathrm{lcm}(1, \ldots, i-1))$. Let $f_i(x)$ be the sum of the first $d - i$ summands of $f(x)$, i.e., $f_i(x) = \alpha_d \cdot (x)_n + \alpha_{n-1} \cdot (x)_{n-1} + \ldots + \alpha_{n-i+1} \cdot (x)_{n-i+1}$.

Initially, the pebble is located on the northernmost tile of column 0. To test whether $\ell = f(h)$, the robot will move the pebble along the northernmost row in phases, until it is eventually shifted $f(h)-1$ steps to the *NE* from its original position. If upon termination the pebble is located at the northernmost tile of column $\ell - 1$, then $f(h) = \ell$.

The algorithm proceeds in phases $d, \ldots, 0$. We maintain the invariant that after phase i for all $i > 0$, the pebble is located at the northernmost tile of column $f_i(h)$. That is, in phase i, the robot moves the pebble $|\alpha_i \cdot (h)_i|$ steps *NE*, if α_i is positive, and *SW*, otherwise. In the final phase $i = 0$, the robot moves the pebble by $|\alpha_0 - 1|$ steps *NE*, if $a_0 \geq 1$, and *SW*, otherwise. For now, assume that each movement can be carried out without moving the pebble outside of the parallelogram. We will later describe how to lift this restriction.

We now describe how the pebble is moved by $|\alpha_i \cdot (h)_i|$ steps. First, note that $\alpha_i \cdot (h)_i = \alpha_i \cdot g_i(h) \cdot \mathrm{lcm}_i(h)$. The first factor α_i is a constant. The second factor $g_i(h)$ can be determined as follows. We encode all possible values of $g_i(\cdot)$ for all $i \in \{0, \ldots, d\}$ into the robot's memory, which can be done since d is constant and $g_i(\cdot)$ has a constant period. Before the main algorithm's execution, the robot can compute $g_i(h)$ for all i by moving through column 0 from north to south: Starting with $g_i(1)$, in every step to the south the robot computes the subsequent function value until the period of $g_i(\cdot)$ is reached, in which case it restarts with $g_i(1)$. When it reaches the southernmost tile of the column, it knows $g_i(h)$ for all i.

We next show how the robot moves the pebble by $\mathrm{lcm}_i(h)$ steps, which, by repeating the movement $|\alpha_i \cdot g_i(h)|$ times, concludes how the complete movement by $|\alpha_i \cdot (h)_i|$ steps is performed. Assume the pebble is in some column c and $\mathrm{lcm}_i(h) \mid c$ (which we will prove by induction shortly). Following the idea of our introductory example, the robot alternates between the following two operations: (1) move the pebble into column c' by moving it one step *NE*, if $\alpha_i > 0$, or *SW*, otherwise; (2) verify whether $\mathrm{lcm}_i(h) \mid c'$ as follows. The robot first performs the zig-zag movement from the proof of Theorem 8.2 to verify whether $h \mid c'$, i.e., whether a *NE* movement moves the robot onto a tile occupied by the pebble. It continues to analogously verify whether $(h-1) \mid c'$, $(h-2) \mid c'$, ..., $(h-i+1) \mid c'$ by performing a modified zig-zag movement an additional $i-1$ times. Here, the zig-zags of the j-th verification are adjusted accordingly by moving j steps to the south prior to each sequence of *SE* movements. The robot stops alternating between the two above operations once the pebble has been moved to a column c' such that $\mathrm{lcm}_i(h) \mid c'$ for the first time. Then, the pebble must have been moved by $\mathrm{lcm}_i(h)$ steps.

It remains to prove that when the robot wants to move the pebble, currently occupying a node of column c, by $\mathrm{lcm}_i(h)$ steps for some i, then $\mathrm{lcm}_i(h) \mid c$. The invariant holds initially for $c = 0$. Now assume that the pebble lies in column c

and the robot wants to move the pebble by $\mathrm{lcm}_i(h)$ steps. By induction hypothesis, $\mathrm{lcm}_i(h) \mid c$. Since $\mathrm{lcm}_j(h) \mid \mathrm{lcm}_i(h)$ for any $j \leq i$, we have that $\mathrm{lcm}_j(h) \mid c \pm \mathrm{lcm}_i(h)$. Therefore, and since the robot can only move the pebble by $\mathrm{lcm}_j(h)$ steps for $j \leq i$ in the next iteration, the induction hypothesis holds for the next iteration as well.

Finally, we show how the robot can resolve *overflows*, i.e., situations in which the algorithm would move the pebble outside of the parallelogram. First, note that the execution of the algorithm after an overflow can, in principle, be continued by the robot by "mirroring" all movements beyond the westernmost or easternmost column, carrying them out into reverse direction. Let h be sufficiently large such that $|\alpha_i \cdot (h)_i + \ldots + \alpha_0| \leq f(h)$ for all i. For all small $h = \mathcal{O}(\max_i(|\alpha_i|))$, we can encode the constantly many possible function values into the robot's state and test them prior to the algorithm's execution by traversing the two sides of the parallelogram once. If throughout the execution of the algorithm the robot ever attempts to move the pebble into a column west of column 0 or east of "virtual" column 2ℓ (while performing the mirroring method from above), it would subsequently not be able to ever move the pebble back into column ℓ (following from the assumption that h is sufficiently large), and consequently $\ell \neq f(h)$. Therefore, the robot can prematurely terminate with a negative result whenever it encounters such a situation. □

The next theorem gives a lower bound on the number of states needed to decide whether $\ell = h^a$, $a \in \mathbb{N}$, thereby proving that no robot with one pebble can decide whether $\ell = f(h)$ for $f(x) = \omega(x^a)\, \forall a \in \mathbb{N}$.

Theorem 8.6 (Polynomial Lower Bound)**.** *A robot with s states and a single pebble cannot decide whether the tile configuration is a parallelogram of height h and length $\ell = f(h)$, $f(x) = \omega(x^{6s+2})$.*

Proof. First, we give a brief outline of the following proof. Assuming such a robot exists, we place the robot with its pebble on the northernmost tile of column 0 of a parallelogram P with height h and length $\ell = f(h)$. We begin by subdividing P horizontally into parallelograms of height h which we will refer to as *blocks*. We define the westernmost and easternmost blocks as the *outer blocks* O_ℓ and O_r, respectively, and denote all other blocks as *inner blocks*. We choose the length b of all inner blocks such that when the robot moves through a sequence of inner blocks (from west to east or from east to west), while either carrying the pebble the entire time or not visiting it at all, its row and state repeat every b columns. The outer blocks will have length at least b.

As in the proof of Theorem 8.4, we consider the execution of the algorithm as a sequence of tuples of nodes and states, and divide it into subsequences $\pi_1, \ldots, \pi_m$. Subsequence π_i starts with the robot carrying the pebble into some outer block, and ends when it reaches the opposite outer block while carrying the pebble. The robot terminates in π_m, and, as this is the final subsequence, before entering the opposite block while carrying the pebble. We define r_{π_i} and q_{π_i} to be the robot's row and state at the beginning of π_i. By considering where the robot places and picks up the pebble, we identify a value d such that r_{π_i} and q_{π_i} remain the same if the robot is executed on a parallelogram P' of height h and length $f(h) + db$, and therefore falsely terminates with a positive result.

Determine Value b We begin the proof by identifying a value b. Consider the robot's execution without the pebble on a parallelogram of height h and infinite length in both directions, starting in row r and state q. As in the proof of Theorem 8.4, we enumerate the parallelogram's rows from 1 to h from N to S, and refer to the empty nodes above and below the parallelogram as rows 0 and $h+1$, respectively. If the robot only moves within at most $s(h+2)$ columns, we define $d_{r,q} := 1$. Otherwise, by the pigeonhole principle, there are two columns that are visited on a node of the same row and in the same state. In this case, we define $d_{r,q}$ to be the distance between these two columns. Note that the robot moves arbitrarily far in this case, its row and state repeating every $d_{r,q}$ columns. Let

$$D := \{d_{r,q} \mid r \in \{0, \dots, h+1\},\, q \in Q\}.$$

Analogously, we consider the execution of the robot if it initially carries the pebble, and define $d^*_{r,q} := 1$, if it ever drops the pebble or only moves within $s(h+2)$ columns. Otherwise, there is a repetition every $d^*_{r,q}$ columns, and the robot carries the pebble indefinitely far. Let

$$D^* := \{d^*_{r,q} \mid r \in \{0, \dots, h+1\},\, q \in Q\}.$$

We now show that $|D| \leq 3s$. If the robot ever is in row r in state q, or row r' in state q', and visits the row 1 or row h in the same state q^* in both executions, the executions will be identical afterwards, and therefore we can choose $d_{r,q} = d_{r',q'}$. Hence, there can only be s combinations of rows and states with distinct executions in which the robot visits the northern- or southernmost row, and these executions contribute at most $2s$ distinct distances to D. Now, let q, r, r' such that the robot does not visit the northern- and southernmost row when starting in state q in row r or r'. Clearly, the robot performs the exact same actions in both executions. Therefore, $d_{r,q} = d_{r',q}$, and these executions contribute at most s additional distances. Therefore, $|D| \leq 3s$. Analogously, we have $|D^*| \leq 3s$.

We set $b := \prod_{\delta \in D \cup D^*} \delta$. If $b \leq s(h+2)$, redefine $b := (s(h+2))b$. Since each distance value is at most $\mathcal{O}(sh)$, and since s is constant, it holds that $b = \mathcal{O}(h^{6s})$. We subdivide P into a maximum number of blocks such that inner blocks have length b and outer blocks have length greater than b.

Determine Value d We now identify a value d. Consider the subsequence π_i, $i < m$. W.l.o.g., assume that the robot moves the pebble from O_ℓ to O_r. Let (r_j, c_j, q_j), $j = 1, \dots, l$ be the row, column and state in which the pebble is dropped during π_i; if it is never dropped, then $l = 0$. We distinguish two cases: In the first case, the robot at some point moves by more than $s(h+2)$ columns while carrying the pebble. In this case, it will move until reaching column $\ell - 1$ due to having a repetition in its state and row. Therefore, whenever the pebble was placed before that, it can only have been previously picked up in a column that is closer than $s(h+2)$ steps, i.e., $|c_{j+1} - c_j| \leq s(h+2)$ for $j < l$. In this case, set $d_i = 1$.

In the second case, the robot does never move by more than $s(h+2)$ columns to the east while carrying the pebble during π_i. Thus, the pebble is dropped within the first (i.e., western) $s(h+2)$ columns of each inner block. P contains $\Omega(f(h)/b) = \omega(h^2) > 2(sh)^2$ inner blocks, for sufficiently large h. For some column c, we denote its index inside its block as $\tilde{c}$. Hence, for each inner block, there exists a j such that c_j is in that block and $\tilde{c}_j < s(h+2)$. There are at most $h \cdot s(h+2) \cdot s \leq 2(sh)^2$ possibilities for $(r_j, \tilde{c}_j, q_j)$. By the pigeonhole principle, there exist $a < b$ such that $c_a < c_b$ and $(r_a, \tilde{c}_a, q_a) = (r_b, \tilde{c}_b, q_b)$. We set $d_i := (c_b - c_a)/b$, i.e., the number of inner blocks between c_a and c_b, and $d := \prod_{i=1}^{m} d_i$.

Execution in P' Let P' be the parallelogram of height h and length $f(h) + db$. Now we show that $(r_{\pi_{i+1}}, q_{\pi_{i+1}})$ is the same in the execution on P and P' for all $i < m$ and that the robot terminates with the same result during π_m. To that end, we will again look at a subsequence $\pi_i,\ i < m$ where, w.l.o.g, the pebble is carried from O_ℓ to O_r and compare the executions on P and P'. Let (r_j, c_j, q_j), $j = 1, \dots, l$ as above. The first l times the pebble is dropped on P' are identical to those on P, since the only difference in the execution between dropping the pebble at a column $c_j,\ j \leq l$, and picking it back up again can be when the robot moves to column $\ell - 1$. As argued above, moving to column $\ell - 1$ on P and P' cannot be distinguished by the robot due to a repetition in row and state and by our choice of b. Thus, the pebble is picked up in the same state again.

Next, we need to look at what happens on P' after the pebble has been picked up for the l-th time. Here, we distinguish between the two cases from above. In the first case, the robot carries the pebble by more than $s(h+2)$ columns to the east on P before entering O_r and, consequently, continues until reaching column $\ell - 1$. The same behavior occurs on P', only that the robot traverses an additional d blocks and, by our choice of b, enters O_r in the same row and state as on P.

In the second case, the robot never carries the pebble by more than $s(h+2)$ columns to the east during π_i. We identified a, b such that $(r_a, \tilde{c}_a, q_a) = (r_b, \tilde{c}_b, q_b)$, i.e., the pebble is picked up in the same row, block-column, and state. Note that after each of these two drops, the robot will return to the pebble (and pick it up) in the same state in P and in P'. This is clearly true if the robot does not visit column 0 and $\ell - 1$ between dropping and placing the pebble in P, which implies that it also does not do so in P'. Otherwise, the robot would have to traverse more than $s(h+2)$ columns without seeing the pebble in P, its row and state repeating every δ columns for some $\delta \in D$. By our choice of b, this implies that the robot returns to the pebble in the same state in P'. Therefore, in both the executions on P and P' the robot will repeatedly drop and pick up the pebble, each repetition occurring d_i blocks to the east from the previous one. As $d_i \mid d$, the robot enters O_r in the same row on P and P' and we thus also have identical $(r_{\pi_{i+1}}, q_{\pi_{i+1}})$.

It only remains to show that the robot terminates with the same result during π_m. W.l.o.g, let π_m begin in O_l on P, and recall that the pebble does not reach O_r before the robot terminates. As argued before, the executions on P and P' can only differ when the robot drops the pebble and moves to the easternmost column of the respective parallelogram. Due to a repetition in state and row, and by our choice of b, the robot again enters the respective easternmost columns in the same state and row, and afterwards picks up the pebble in the same state on both P and P'. Therefore, the robot ultimately terminates in the same state in P and P', proving the theorem. $\square$

8.2.3. A Robot With Two Pebbles

Next, we show that having two pebbles enables the robot to decide certain exponential functions. Note that by Theorem 8.6, the following result is optimal in the number of pebbles used.

Theorem 8.7 (Power Tower)**.** *A robot with two pebbles can decide whether a given tile configuration is a parallelogram with height $h > 1$ and length*

$$\ell = 2^{2^{\cdot^{\cdot^{\cdot^{2^h}}}}}, \text{ where the power tower is of constant height.}$$

Proof. Let $d+1$ be the height of the power tower (i.e., there are d twos in the function) and denote the two pebbles as a and b. Pebble a always resides in the northernmost row of the parallelogram. We use a's column index as a register on which we perform basic arithmetic operations using b as a helper. Note that although the easternmost column of the parallelogram has index $\ell - 1$, we describe the algorithm as if there was an additional column ℓ. A movement from or to column ℓ can easily be simulated by the robot.

The algorithm is divided into three *stages*. The purpose of the first stage is to verify that ℓ is a power of 2. In the second stage, we move a from column ℓ to column $\log^{(d-1)}(\ell)$, where $\log^{(d-1)}(\cdot)$ denotes the application of the logarithm $d-1$ times. In the third stage, we verify that a is in column 2^h after the second stage. If in any of the three stages the robot detects a violation of any assumption, i.e., if according to the algorithm a would have to be moved beyond column 0 or ℓ, or should be in column 0 or ℓ, but is not, the robot terminates with a negative result.

Before we describe the three stages in more detail, we describe how a's column index can be multiplied or divided by any constant $c \geq 1$, provided that the result is integer and between 0 and ℓ. We first place b at a tile south of a, and then move a into column 0. In case of a multiplication, we then alternatingly move a *NE* by c steps, and b *SW* by one step, until b reaches column 0. In case of a division, we correspondingly move b by c steps and a by one step.

Stage 1 In the first stage, the robot does the following. It first places a at the northernmost tile of column 1. It then repeatedly multiplies by 2 (i.e., multiplies a's column index by 2) using the above-mentioned strategy. If b reaches column 0 immediately after a has reached column ℓ, ℓ is a power of 2.

Stage 2 At the beginning of the second stage, a is placed at the northernmost tile of column ℓ. The stage is divided into $d-1$ phases, where in each phase a is moved from column i to column $\log(i)$. Note that since d is a constant, the robot can count the number of phases. Furthermore, after the first phase ℓ is verified to be a power of 2, and, as we will show later, if a's column index is 2^x at the beginning of a phase, but x is not a power of 2, then the phase fails. Therefore, a's column index is ensured to be a power of 2 at the beginning of each phase.

In each phase, a is moved from some column 2^x to column x in a step-wise fashion. More precisely, in the j-th step it is moved from column $5^{j-1} \cdot 2^{x/2^{j-1}}$ to column $5^j \cdot 2^{x/2^j}$. This is continued until the resulting column index is not divisible by 4 anymore, i.e., when it becomes $5^{\log x} \cdot 2$. The general idea is to use the exponent of 5 as a counter on the number of times x needs to be divided by 2 until it becomes 1, which yields its logarithm. This idea is based on [Min67, Section 14].

It remains to show how a single step can be performed. First, the robot moves a from column $5^{j-1} \cdot 2^{x/2^{j-1}}$ into column $5^{j-1} \cdot 3^{x/2^j}$ by alternatingly dividing by 4 and multiplying by 3. After each repetition, the robot verifies whether a's column index is divisible by 2. This is done by traversing the northernmost row from west to east until a is reached, and counting the number of steps modulo 2. Note that if $x/2^{j-1}$ is even, which is the case if x is a power of 2, then the division by 4 is always possible. Otherwise, the division by 4 will fail at latest when $x/2^{j-1}$ becomes 1. Once a's column index is not divisible by 2 anymore, a is in column $5^{j-1} \cdot 3^{x/2^j}$.

Analogously, by alternatingly dividing a's column index by 3 and multiplying by 2 as long as the column index is divisible by 3, the robot afterwards moves a into column $5^{j-1} \cdot 2^{x/2^j}$. By multiplying with 5, a is finally moved into column $5^j \cdot 2^{x/2^j}$.

After each step, the robot verifies whether a's column index is divisible by 4 (i.e., if $x/2^j \geq 2$). If so, it continues with the next step. Otherwise, $x/2^j = 1$, and thus $j = \log x$ and a must be in column $5^{\log x} \cdot 2$. From there, a can easily be moved into column x by first dividing by 2, and afterwards alternatingly dividing by 5 and multiplying by 2 until the column index is not divisible by 5 anymore.

Note that if a's column index is 2^x at the beginning of some phase, but x is not a power of 2, then at some step $x/2^{j-1}$ will be odd, and, as described above, the algorithm will fail. Further note that for $h \geq 8$, which can be verified beforehand, moving a from column 2^x (where, consequently, x must be at least 8) to $5^{\log x} \cdot 2$ can only decrease a's column index. Therefore, although a might be moved *NE* in the final steps of a phase, it can easily be seen it will be moved sufficiently far *SW* within the first steps such that it is never moved beyond column ℓ. If that happens nonetheless, the robot terminates with a negative result.

Stage 3 Finally, in the third stage, the robot verifies that a is in column 2^h by diving by 2 for h times. To count up to h, b initially resides in row 2 and is moved one step S after each division. When b reaches a southernmost tile, a must lie in column 4, which can easily be verified by the robot. ☐

The algorithm of the previous theorem can also be adapted for any power tower whose height is a linear function $\alpha h + \beta$ for constants $\alpha, \beta \in \mathbb{N}$. To count the number of phases in the second stage, we move a south after every α phases, and, after having reached a southernmost tile, finally count β additional phases. The highest exponent of the power tower may also be a function linear in h, which can be handled correspondingly in the third stage.

The algorithm can further be adapted for any other base $c \in \mathbb{N}$. Note that if c is a composite number, we need to apply the operations to its prime factors separately, taking into account their powers. Since it is well-known that for $n \geq 25$ there is always a prime between n and $(1+1/5)n$ [Nag52], we can use the two smallest primes that are no prime factors of c instead of 3 and 5 in the second stage of the algorithm. We conclude the following corollary.

Corollary 8.8 (Linear Power Tower)**.** *A robot with two pebbles can decide whether a given tile configuration is a parallelogram with height $h > 1$ and length*

$\ell = c^{c^{\cdot^{\cdot^{\cdot^{c^h}}}}}$*, where the power tower is of height $\alpha h + \beta$, and $\alpha, \beta, c \in \mathbb{N}_0$ are constant.*

8.2.4. A Family of Functions Requiring an Increasing Amount of Pebbles

The discussion from the previous sections naturally brings up the question whether there is a function family whose detection requires an increasing amount of pebbles. In this section, we answer that question positively. As the proofs are fairly straightforward, we mostly state the general ideas, leaving out some details.

In order to simplify analysis, we consider a robot with pebbles operating on a line segment instead of a parallelogram. We also assume that pebbles are distinguishable and can be placed onto the same tile. Note that it is easy to simulate colors of a finite amount of pebbles on a line segment by keeping track of the order of the pebbles. Furthermore, the robot can simulate placing two pebbles onto each other by placing the second pebble within constant distance instead and save the offset from its intended position.

First, we introduce the notion of *program machines* [Min67, Section 11].

Definition 8.9 (Program Machine)**.** *A* program machine *consists of a finite set of registers, holding arbitrary numbers from $\mathbb{N}_0$, and a finite program of numbered instructions from the following instruction set.*

- *zero(r): Set register $r \leftarrow 0$.*
- *increment(r): Set register $r \leftarrow r + 1$.*
- *decrementOrJump(r, k): If $r = 0$, jump to instruction k. Otherwise, set $r \leftarrow r - 1$.*
- *halt: Stop the execution.*

It is well-known that using the instructions of Definition 8.9, it is possible to simulate instructions to copy the value of register r_1 to r_2, and to jump if (or unless) $r_1 = r_2$ [Min67].

Lemma 8.10. *A program machine with* 3 *registers can simulate a deterministic Turing machine with $\Gamma = \Sigma = \{0, 1\}$.*

Proof. It is well known that a Turing machine can be simulated using two stacks containing the tape's bits behind and in front of the head, respectively. These stacks can be viewed as the binary encoding of natural numbers. They will be stored in registers one and two.

Using the third register as a scratch pad, doubling and halving a register is simple. To pop a bit from the stack, we divide by two and look at the remainder. To push a bit onto the stack, we multiply by two and add the bit. For more details, we refer the reader to [Min67, Section 11]. □

Since we are interested in a robot moving on a finite space, we now define *bounded program machines.*

Definition 8.11 (Bounded Program Machine)**.** *A* bounded program machine *is a program machine whose first register is initialized to the input x. The other registers are initialized to 0. No register may exceed x.*

The following observation follows from the fact that a robot can easily simulate a bounded program machine using pebbles, and vice versa.

Observation 8.12 (Pebble Simulation)**.** *The computational power of a robot with k pebbles on a line of length $x \geq k$ is between that of a k and a $k+1$ register bounded program machine with input x.*

The next two lemmas show that deterministic linear bounded automata are essentially equivalent in their computational capabilities to bounded program machines and that the number of registers relates to the number of tape symbols needed. This enables us to apply well-known results from complexity theory to our model. We use the definition of *deterministic linear bounded automata* (LBA) from [FO73] as Turing machines that never leave the cells in which their input was placed. Inputs are restricted to $\{0,1\}^*$.

Lemma 8.13 (LBA Simulation with Bounded Program Machine)**.** *A deterministic linear bounded automaton with $|\Gamma| = k$ and $\Sigma = \{0,1\}$ with input $x \in \Sigma^*$ can be simulated using a bounded program machine with $1+2\lceil\log k\rceil$ registers and with input $x' := (1x)_2$.*

Proof. The construction from Lemma 8.10 ensures that no register will be increased beyond x' when simulating the linear bounded automaton. The two stacks will now contain elements in Γ. We encode the symbols in binary and store their representation using $2\lceil\log k\rceil$ registers. □

Lemma 8.14 (Bounded Program Machine Simulation with LBA)**.** *A k register bounded program machine with input $(1x)_2$, $x \in \{0,1\}^*$, can be simulated by a deterministic linear bounded automaton with 2^k tape symbols and input x.*

Proof. The tape stores the binary representation of the registers' values, each symbol representing one bit of k registers. Operations from Definition 8.9 are now trivial to perform. □

Finally, it can be shown that there exists a family of languages requiring deterministic linear bounded automata with an increasing amount of tape symbols. These will directly translate to sets of accepted line lengths.

Lemma 8.15 ([FO73, Corollary 2])**.** *There exists a family of context sensitive languages $S_k \subseteq \{0,1\}^*$ accepted by some deterministic linear bounded automaton, but not using fewer than k symbols.*

Together with Observation 8.12, the previous lemmas imply the following theorem.

Theorem 8.16 (Pebble Lower Bound). *There exists a family $L_k \subseteq \mathbb{N}$ such that a robot exists that can detect whether a line has length $\ell \in L_k$ using a finite number of pebbles but none using less than k pebbles.*

To construct a family of functions for deciding side ratios of a parallelogram, we can set

$$f_k(h) = \begin{cases} 1, & h \in L_k \\ 2, & h \notin L_k. \end{cases}$$

Note that the resulting parallelogram differs from the previous examples in that its length is only 1 or 2. The robot can perform the simulation of the bounded program machine using the parallelogram's height, and a length of 2 does not give it too much power. Furthermore, we can modify the function family such that $\ell \geq h$ while still requiring k pebbles to decide $f_k(h)$.

8.3. Outlook

In this chapter, we have identified some simple functions that can or cannot be decided with a given set of pebbles. Beyond our study in Section 8.2.4, we are interested to find functions, such as superexponentials, that require more than only two pebbles. Furthermore, it is an interesting question whether, instead or in addition to using pebbles, multiple robots can help in shape recognition problems. For example, it is still an open question whether two robots, which are activated in an arbitrary order, are more powerful than a single robot with a pebble. Moreover, there are many different model assumptions under which our problems can be investigated.

Whereas we primarily focused on detecting parallelograms of certain side ratios, we are very interested to examine whether our results and algorithms are applicable to other shapes as well. For example, it may be possible to recognize more complex structures such as irregular hexagons with certain side ratios, or to even come up with a more general procedure to recognize larger families of shapes. Furthermore, rasterized disks or ellipses might be interesting shapes to consider. Other intriguing problems related to shape recognition include testing whether a structure is symmetric or simply connected.

Bibliography

[Abd+20] A. Abdel-Rahman, A. T. Becker, D. E. Biediger, K. C. Cheung, S. P. Fekete, N. A. Gershenfeld, S. Hugo, B. Jenett, P. Keldenich, E. Niehs, C. Rieck, A. Schmidt, C. Scheffer, and M. Yannuzzi. "Space Ants: Constructing and Reconfiguring Large-Scale Structures with Finite Automata". In: *36th International Symposium on Computational Geometry (SoCG)*. Vol. 164. 2020. DOI: 10.4230/LIPIcs.SoCG.2020.73.

[ABI86] N. Alon, L. Babai, and A. Itai. "A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem". In: *Journal of Algorithms* 7.4 (1986), pp. 567–583. DOI: 10.1016/0196-6774(86)90019-2.

[ACK16] A. Abboud, K. Censor-Hillel, and S. Khoury. "Near-Linear Lower Bounds for Distributed Distance Computations, Even in Sparse Networks". In: *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*. 2016, pp. 29–42. DOI: 10.1007/978-3-662-53426-7_3.

[Afe+90] Y. Afek, G. M. Landau, B. Schieber, and M. Yung. "The Power of Multimedia: Combining Point-to-Point and Multiaccess Networks". In: *Information and Computation* 84.1 (1990), pp. 97–118. DOI: 10.1016/0890-5401(90)90035-G.

[Ale82] R. Aleliunas. "Randomized Parallel Communication". In: *Proceedings of the 1st ACM Symposium on Principles of Distributed Computing (PODC)*. 1982, pp. 60–72. DOI: 10.1145/800220.806683.

[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)*. 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.

[AS07] J. Aspnes and G. Shah. "Skip graphs". In: *ACM Transactions on Algorithms (TALG)* 3.4 (2007). DOI: 10.1145/1290672.1290674.

[AS18] J. Augustine and S. Sivasubramaniam. "Spartan: A Framework For Sparse Robust Addressable Networks". In: *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 1060–1069. DOI: 10.1109/IPDPS.2018.00115.

[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 IEEE Annual Symposium on Foundations of Computer Science (FOCS).* 2015, pp. 350–369. DOI: 10.1109/FOCS.2015.29.

[Aug+19] J. Augustine, M. Ghaffari, R. Gmyr, K. Hinnenthal, F. Kuhn, J. Li, and C. Scheideler. "Distributed Computation in Node-Capacitated Networks". In: *Proceedings of the 31st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 2019, pp. 69–79. DOI: 10.1145/3323165.3323195.

[Aug+20a] J. Augustine, K. Choudhary, A. Cohen, D. Peleg, S. Sivasubramaniam, and S. Sourav. "Distributed Graph Realizations". In: *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 2020, pp. 158–167. DOI: 10.1109/IPDPS47924.2020.00026.

[Aug+20b] J. Augustine, K. Hinnenthal, F. Kuhn, C. Scheideler, and P. Schneider. "Shortest Paths in a Hybrid Network Model". In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* 2020, pp. 1280–1299. DOI: 10.1137/1.9781611975994.78.

[AV84] M. 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.

[AW07] J. Aspnes and Y. Wu. "$O(\log n)$-Time Overlay Network Construction from Graphs with Out-Degree 1". In: *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS).* 2007, pp. 286–300. DOI: 10.1007/978-3-540-77096-1_21.

[Awe87] B. Awerbuch. "Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election, and related problems". In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC).* 1987, pp. 230–240. DOI: 10.1145/28395.28421.

[Bar+16] L. Barenboim, M. Elkin, S. Pettie, and J. Schneider. "The Locality of Distributed Symmetry Breaking". In: *Journal of the ACM* 63.3 (2016), 20:1–20:45. DOI: 10.1145/2903137.

[BE10] L. Barenboim and M. Elkin. "Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition". In: *Distributed Computing* 22.5-6 (2010), pp. 363–379. DOI: 10.1007/s00446-009-0088-2.

[BE11] L. Barenboim and M. Elkin. "Deterministic Distributed Vertex Coloring in Polylogarithmic Time". In: *Journal of the ACM (JACM)* 58.5 (2011), pp. 1–25. DOI: 10.1145/2027216.2027221.

[Bec+17] R. Becker, A. Karrenbauer, S. Krinninger, and C. Lenzen. “Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models”. In: *Proceedings of the 31st International Symposium on Distributed Computing (DISC).* 2017, 7:1–7:16. DOI: `10.4230/LIPIcs.DISC.2017.7`.

[BEK14] L. Barenboim, M. Elkin, and F. Kuhn. “Distributed $(\Delta + 1)$-Coloring in Linear (in Δ) Time”. In: *SIAM Journal on Computing* 43.1 (2014), pp. 72–95. DOI: `10.1137/12088848X`.

[Bel58] R. Bellman. “On a routing problem”. In: *Quarterly of Applied Mathematics* 16.1 (1958), pp. 87–90. DOI: `10.1090/qam/102435`.

[Ben+12] B. Ben-Moshe, A. Dvir, M. Segal, and A. Tamir. “Centdian Computation in Cactus Graphs”. In: *Journal of Graph Algorithms and Applications* 16.2 (2012), pp. 199–224. DOI: `10.7155/jgaa.00255`.

[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`.

[BH17] B. Brimkov and I. V. Hicks. “Memory efficient algorithms for cactus graphs and block graphs”. In: *Discrete Applied Mathematics* 216 (2017), pp. 393–407. DOI: `10.1016/j.dam.2015.10.032`.

[BK18] L. Barenboim and V. Khazanov. “Distributed Symmetry-Breaking Algorithms for Congested Cliques”. In: *International Computer Science Symposium in Russia.* 2018, pp. 41–52. DOI: `10.1007/978-3-319-90530-3_5`.

[BK78] M. Blum and D. Kozen. “On the Power of the Compass (or, Why Mazes are Easier to Search than Graphs)”. In: *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS).* 1978, pp. 132–142. DOI: `10.1109/SFCS.1978.30`.

[Ble+14] G. E. Blelloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tangwongsan. “Nearly-Linear Work Parallel SDD Solvers, Low-Diameter Decomposition, and Low-Stretch Subgraphs”. In: *Theory of Computing Systems* 55.3 (2014), pp. 521–554. DOI: `10.1007/s00224-013-9444-5`.

[BN11] A. Bonato and R. J. Nowakowski. *The Game of Cops and Robbers on Graphs.* AMS, 2011. ISBN: 978-0-8218-5347-4.

[BS07] S. Baswana and S. Sen. “A Simple and Linear Time Randomized Algorithm for Computing Sparse Spanners in Weighted Graphs”. In: *Random Structures & Algorithms* 30.4 (2007), pp. 532–563. DOI: `10.1002/rsa.20130`.

[Bud78] L. Budach. “Automata and Labyrinths”. In: *Mathematische Nachrichten* 86.1 (1978), pp. 195–282. DOI: `10.1002/mana.19780860120`.

[Cel+13] L. E. Celis, O. Reingold, G. Segev, and U. Wieder. “Balls into Bins: Smaller Hash Families and Faster Evaluation”. In: *SIAM Journal on Computing* 42.3 (2013), pp. 1030–1050. DOI: 10.1137/120871626.

[Cen+19a] K. Censor-Hillel, M. Dory, J. H. Korhonen, and D. Leitersdorf. “Fast Approximate Shortest Paths in the Congested Clique”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC).* 2019, pp. 74–83. DOI: 10.1145/3293611.3331633.

[Cen+19b] K. Censor-Hillel, P. Kaski, J. H. Korhonen, C. Lenzen, A. Paz, and J. Suomela. “Algebraic methods in the congested clique”. In: *Distributed Computing* 32.6 (2019), pp. 461–478. DOI: 10.1007/s00446-016-0270-2.

[CKS20] J. Castenow, C. Kolb, and C. Scheideler. “A Bounding Box Overlay for Competitive Routing in Hybrid Communication Networks”. In: *Proceedings of the 21st International Conference on Distributed Computing and Networking (ICDCN).* 2020, pp. 1–10. DOI: 10.1145/3369740.3369777.

[CLP20] K. Censor-Hillel, D. Leitersdorf, and V. Polosukhin. *Distance Computations in the Hybrid Network Model via Oracle Simulations.* 2020. arXiv: 2010.13831 [cs.DC].

[CLT20] K. Censor-Hillel, D. Leitersdorf, and E. Turner. “Sparse matrix multiplication and triangle listing in the Congested Clique model”. In: *Theoretical Computer Science* 809 (2020), pp. 45–60. DOI: 10.1016/j.tcs.2019.11.006.

[CPS20] K. Censor-Hillel, M. Parter, and G. Schwartzman. “Derandomizing local distributed algorithms under bandwidth restrictions”. In: *Distributed Computing* 33.3 (2020), pp. 349–366. DOI: 10.1007/s00446-020-00376-1.

[Das+12] A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. “Distributed Verification and Hardness of Distributed Approximation”. In: *SIAM Journal on Computing* 41.5 (2012), pp. 1235–1265. DOI: 10.1137/11085178X.

[Das13] S. Das. “Mobile Agents in Distributed Computing: Network Exploration”. In: *Bulletin of the European Association for Theoretical Computer Science* 109 (2013), pp. 54–69.

[Day+17] J. J. Daymude, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. “Improved leader election for self-organizing programmable matter”. In: *Algorithms for Sensor Systems - International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS).* 2017, pp. 127–140. DOI: 10.1007/978-3-319-72751-6_10.

[Day+18] J. J. Daymude, Z. Derakhshandeh, R. Gmyr, A. Porter, A. W. Richa, C. Scheideler, and T. Strothmann. "On the runtime of universal coating for programmable matter". In: *Natural Computing* 17.1 (2018), pp. 81–96. DOI: 10.1007/s11047-017-9658-6.

[Day+19] J. J. Daymude, K. Hinnenthal, A. W. Richa, and C. Scheideler. "Computing by Programmable Particles". In: *Distributed Computing by Mobile Entities: Current Research in Moving and Computing.* 2019, pp. 615–681. DOI: ComputingbyProgrammableParticles.

[Day+20] J. J. Daymude, R. Gmyr, K. Hinnenthal, I. Kostitsyna, C. Scheideler, and A. W. Richa. "Convex Hull Formation for Programmable Matter". In: *Proceedings of the 21st International Conference on Distributed Computing and Networking (ICDCN).* 2020, pp. 1–10. DOI: 10.1145/3369740.3372916.

[Dem+03] E. D. Demaine, M. L. Demaine, M. Hoffmann, and J. O'Rourke. "Pushing blocks is hard". In: *Computational Geometry* 26.1 (2003), pp. 21–36. DOI: 10.1016/S0925-7721(02)00170-0.

[Der+14] Z. Derakhshandeh, S. Dolev, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "Amoebot—A New Model for Programmable Matter". In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 2014, pp. 220–222. DOI: 10.1145/2612669.2612712.

[Der+16] 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).* 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).* 2016, pp. 417–427. DOI: 10.1145/2935764.2935783.

[Di +20] G. A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, and Y. Yamauchi. "Shape formation by programmable particles". In: *Distributed Computing* 33.1 (2020), pp. 69–101. DOI: 10.1007/s00446-019-00350-6.

[Dij59] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. DOI: 10.1007/BF01386390.

[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)*. 2014, pp. 367–376. DOI: 10.1145/2611462.2611493.

[DL98] N. Deo and B. Litow. “A Structural Approach to Graph Compression”. In: *Proceedings of the 23th MFCS Workshop on Communications*. 1998, pp. 91–100.

[DLP12] D. Dolev, C. Lenzen, and S. Peled. ““Tri, Tri again”: Finding Triangles and Small Subgraphs in a Distributed Setting”. In: *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*. 2012, pp. 195–209. DOI: 10.1007/978-3-642-33651-5_14.

[DP20] M. Dory and M. Parter. “Exponentially Faster Shortest Paths in the Congested Clique”. In: *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC)*. 2020, pp. 59–68. DOI: 10.1145/3382734.3405711.

[DPZ91] H. N. Djidjev, G. E. Pantziou, and C. D. Zaroliagis. “Computing Shortest Paths and Distances in Planar Graphs”. In: *Proceedings of the 18th International Colloquium International Colloquium on Automata, Languages, and Programming (ICALP)*. 1991, pp. 327–338. DOI: 10.1007/3-540-54233-7_145.

[DW07] V. Dujmovic and D. R. Wood. “Graph Treewidth and Geometric Thickness Parameters”. In: *Discrete & Computational Geometry* 37.4 (2007), pp. 641–670. DOI: 10.1007/s00454-007-1318-7.

[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.

[EN18] M. Elkin and O. Neiman. “Efficient algorithms for constructing very sparse spanners and emulators”. In: *ACM Transactions on Algorithms (TALG)* 15.1 (2018), pp. 1–29. DOI: 10.1145/3274651.

[EW13] C. Evans and E. Winfree. “DNA Sticky End Design and Assignment for Robust Algorithmic Self-Assembly”. In: *Proceedings of DNA Computing and Molecular Computing (DNA)*. 2013, pp. 61–75. DOI: 10.1007/978-3-319-01928-4_5.

[Far+10] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. “Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers”. In: *Proceedings of the ACM SIGCOMM 2010 Conference*. 2010, pp. 339–350. DOI: 10.1145/1851182.1851223.

[Fek+20] S. P. Fekete, E. Niehs, C. Scheffer, and A. Schmidt. “Connected Reconfiguration of Lattice-Based Cellular Structures by Finite-Memory Robots”. In: *Algorithms for Sensor Systems - International Symposium on Algorithms and Experiments for Sensor Systems, Wireless*

Networks and Distributed Robotics (ALGOSENSORS). 2020, pp. 60–75. DOI: 10.1007/978-3-030-62401-9_5.

[Fek+21] S. P. Fekete, R. Gmyr, S. Hugo, P. Keldenich, C. Scheffer, and A. Schmidt. "CADbots: Algorithmic Aspects of Manipulating Programmable Matter with Finite Automata". In: *Algorithmica* 83.1 (2021), pp. 387–412. DOI: 10.1007/s00453-020-00761-z.

[FHS20] M. Feldmann, K. Hinnenthal, and C. Scheideler. "Fast Hybrid Network Algorithms for Shortest Paths in Sparse Graphs". In: *Proceedings of the 24th International Conference on Principles of Distributed Systems (OPODIS)*. 2020, 31:1–31:16. DOI: 10.4230/LIPIcs.OPODIS.2020.31.

[FHW12] S. Frischknecht, S. Holzer, and R. Wattenhofer. "Networks Cannot Compute Their Diameter in Sublinear Time". In: *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2012, pp. 1150–1162. DOI: 10.1137/1.9781611973099.91.

[FN18] S. Forster and D. Nanongkai. "A Faster Distributed Single-Source Shortest Paths Algorithm". In: *Proceedings of the 59th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 2018, pp. 686–697. DOI: 10.1109/FOCS.2018.00071.

[FO73] E. D. Feldman and J. C. Owings. "A class of universal linear bounded automata". In: *Information Sciences* 6 (1973), pp. 187–190. DOI: 10.1016/0020-0255(73)90036-4.

[For56] L. R. Ford. *Network Flow Theory*. Tech. rep. P-923. The Rand Corporation, 1956.

[Fra+05] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. "Graph exploration by a finite automaton". In: *Theoretical Computer Science* 345.2-3 (2005), pp. 331–344. DOI: 10.1016/j.tcs.2005.07.014.

[FS19] K.-T. Foerster and S. Schmid. "Survey of Reconfigurable Data Center Networks: Enablers, Algorithms, Complexity". In: *SIGACT News* 50.2 (2019), pp. 62–79. DOI: 10.1145/3351452.3351464.

[FSS20] M. Feldmann, C. Scheideler, and S. Schmid. "Survey on Algorithms for Self-Stabilizing Overlay Networks". In: *ACM Computing Surveys* 53.4 (2020). DOI: 10.1145/3397190.

[FT08] F. V. Fomin and D. M. Thilikos. "An annotated bibliography on guaranteed graph searching". In: *Theoretical Computer Science* 399.3 (2008), pp. 236–245. DOI: 10.1016/j.tcs.2008.02.040.

[Gha+18] M. Ghaffari, T. Gouleakis, C. Konrad, S. Mitrović, and R. Rubinfeld. "Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover". In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*. 2018, pp. 129–138. DOI: 10.1145/3212734.3212743.

[Gha16] M. Ghaffari. "An Improved Distributed Algorithm for Maximal Independent Set". In: *Proceedings of the 2016 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2016, pp. 270–277. DOI: `10.1137/1.9781611974331.ch20`.

[GHS19] T. Götte, K. Hinnenthal, and C. Scheideler. "Faster Construction of Overlay Networks". In: *International Colloquium on Structural Information and Communication Complexity (SIROCCO)*. 2019, pp. 262–276. DOI: `10.1007/978-3-030-24922-9_18`.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. "A Distributed Algorithm for Minimum-Weight Spanning Trees". In: *ACM Transactions on Programming Languages and systems (TOPLAS)* 5.1 (1983), pp. 66–77. DOI: `10.1145/357195.357200`.

[GKP98] J. A. Garay, S. Kutten, and D. Peleg. "A SubLinear Time Distributed Algorithm for Minimum-Weight Spanning Trees". In: *SIAM Journal on Computing* 27.1 (1998), pp. 302–316. DOI: `10.1137/S0097539794261118`.

[GKS17] M. Ghaffari, F. Kuhn, and H.-H. Su. "Distributed MST and Routing in Almost Mixing Time". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 2017, pp. 131–140. DOI: `10.1145/3087801.3087827`.

[GL18] M. Ghaffari and J. Li. "Improved Distributed Algorithms for Exact Shortest Paths". In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. 2018, pp. 431–444. DOI: `10.1145/3188745.3188948`.

[Gmy+17a] 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)*. 2017, 137:1–137:15. DOI: `10.4230/LIPIcs.ICALP.2017.137`.

[Gmy+17b] R. Gmyr, I. Kostitsyna, F. Kuhn, C. Scheideler, and T. Strothmann. *Forming Tile Shapes with a Single Robot.* Presented at the European Workshop on Computational Geometry (EuroCG). 2017.

[Gmy+18a] R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, and C. Scheideler. "Shape Recognition by a Finite Automaton Robot". In: *Proceedings of the 43rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*. 2018, 52:1–52:15. DOI: `10.4230/LIPIcs.MFCS.2018.52`.

[Gmy+18b] R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, and C. Scheideler. *Shape Recognition by a Finite Automaton Robot.* Presented at the European Workshop on Computational Geometry (EuroCG). 2018.

[Gmy+18c] R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, C. Scheideler, and T. Strothmann. “Forming Tile Shapes with Simple Robots”. In: *Proceedings of DNA Computing and Molecular Programming (DNA).* 2018, pp. 122–138. DOI: 10.1007/978-3-030-00030-1_8.

[Gmy+20] R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, C. Scheideler, and T. Strothmann. “Forming tile shapes with simple robots”. In: *Natural Computing* 19.2 (2020), pp. 375–390. DOI: 10.1007/s11047-019-09774-2.

[Göt+20] T. Götte, K. Hinnenthal, C. Scheideler, and J. Werthmann. *Time-Optimal Construction of Overlay Networks.* 2020. arXiv: 2009.03987.

[GP16] M. Ghaffari and M. Parter. “MST in Log-Star Rounds of Congested Clique”. In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC).* 2016, pp. 19–28. DOI: 10.1145/2933057.2933103.

[GS19] M. Ghaffari and A. Sayyadi. “Distributed Arboricity-Dependent Graph Coloring via All-to-All Communication”. In: *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP).* 2019, 142:1–142:14. DOI: 10.4230/LIPIcs.ICALP.2019.142.

[GVS19] T. Götte, V. R. Vijayalakshmi, and C. Scheideler. “Always be Two Steps Ahead of Your Enemy”. In: *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 2019. DOI: 10.1109/IPDPS.2019.00114.

[GZ05] J. Gao and L. Zhang. “Well-Separated Pair Decomposition for the Unit-Disk Graph Metric and Its Applications”. In: *SIAM Journal on Computing* 35.1 (2005), pp. 151–169. DOI: 10.1137/S0097539703436357.

[Hal+11] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. “Augmenting Data Center Networks with Multi-Gigabit Wireless Links”. In: *Proceedings of the ACM SIGCOMM 2011 Conference.* 2011, pp. 38–49. DOI: 10.1145/2043164.2018442.

[Ham+14] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer. “FireFly: A Reconfigurable Wireless Data Center Fabric Using Free-Space Optics”. In: *Proceedings of the 2014 ACM Conference on SIGCOMM.* 2014, pp. 319–330. DOI: 10.1145/2740070.2626328.

[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 Computing (PODC).* 2015, pp. 91–100. DOI: 10.1145/2767386.2767434.

[Hof81] F. Hoffmann. “One pebble does not suffice to search plane labyrinths”. In: *Proceedings of the International Fundamentals of Computation Theory Conference (FCT).* 1981, pp. 433–444. DOI: `10.1007/3-540-10854-8_47`.

[HPS14] J. W. Hegeman, S. V. Pemmaraju, and V. B. Sardeshmukh. “Near-Constant-Time Distributed Algorithms on a Congested Clique”. In: *Proceedings of the 28th International Symposium on Distributed Computing (DISC).* 2014, pp. 514–530. DOI: `10.1007/978-3-662-45174-8_35`.

[Hur+15] F. Hurtado, E. Molina, S. Ramaswami, and V. Sacristán. “Distributed reconfiguration of 2D lattice-based modular robotic systems”. In: *Autonomous Robots* 38.4 (2015), pp. 383–413. DOI: `10.1007/s10514-015-9421-8`.

[HW12] S. Holzer and R. Wattenhofer. “Optimal Distributed All Pairs Shortest Paths and Applications”. In: *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing (PODC).* 2012, pp. 355–364. DOI: `10.1145/2332432.2332504`.

[HY08] S. Hong and Y. Yang. “On the periodicity of an arithmetical function”. In: *Comptes Rendus Mathematique* 346.13-14 (2008), pp. 717–721. DOI: `10.1016/j.crma.2008.05.019`.

[II86] A. Israeli and A. Itai. “A fast and simple randomized parallel algorithm for maximal matching”. In: *Information Processing Letters* 22.2 (1986), pp. 77–80. DOI: `10.1016/0020-0190(86)90144-4`.

[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`.

[Jac+14] R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig. “SKIP+: A Self-Stabilizing Skip Graph”. In: *Journal of the ACM* 61.6 (2014), 36:1–36:26. DOI: `10.1145/2629695`.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Vol. 17. Addison Wesley, 1992. ISBN: 978-0201548563.

[JM95] D. B. Johnson and P. 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`.

[JN18] T. Jurdziński and K. Nowicki. “MST in $O(1)$ Rounds of Congested Clique”. In: *Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* 2018, pp. 2620–2632. DOI: `10.1137/1.9781611975031.167`.

[Jun+18] D. Jung, C. Kolb, C. Scheideler, and J. Sundermeier. "Competitive Routing in Hybrid Communication Networks". In: *Algorithms for Sensor Systems - International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS).* Vol. 11410. 2018, pp. 15–31. DOI: 10.1007/978-3-030-14094-6_2.

[Kap+18] H. Kaplan, W. Mulzer, L. Roditty, and P. Seiferth. "Routing in Unit Disk Graphs". In: *Algorithmica* 80.3 (2018), pp. 830–848. DOI: 10.1007/s00453-017-0308-2.

[KKT15] V. King, S. Kutten, and M. Thorup. "Construction and Impromptu Repair of an MST in a Distributed Network with $o(m)$ Communication". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC).* 2015, pp. 71–80. DOI: 10.1145/2767386.2767405.

[KMW04] F. Kuhn, T. Moscibroda, and R. Wattenhofer. "What Cannot Be Computed Locally!" In: *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC).* 2004, pp. 300–309. DOI: 10.1145/1011767.1011811.

[Kot+06] K. Kothapalli, M. Onus, C. Scheideler, and C. Schindelhauer. "Distributed Coloring in $\tilde{O}(\sqrt{\log n})$ Bit Rounds". In: *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 2006. DOI: 10.5555/1898953.1898978.

[KP98] S. Kutten and D. Peleg. "Fast Distributed Construction of Small k-Dominating Sets and Applications". In: *Journal of Algorithms* 28.1 (1998), pp. 40–66. DOI: 10.1006/jagm.1998.0929.

[Kru56] J. B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50. DOI: 10.1090/S0002-9939-1956-0078686-7.

[KS17] J. H. Korhonen and J. Suomela. "Brief Announcement: Towards a Complexity Theory for the Congested Clique". In: *Proceedings of the 31st International Symposium on Distributed Computing (DISC).* 2017, 55:1–55:3. DOI: 10.4230/LIPIcs.DISC.2017.55.

[KS18] U. N. Kar and D. K. Sanyal. "An overview of device-to-device communication in cellular networks". In: *ICT Express* 4.3 (2018), pp. 203–208. DOI: 10.1016/j.icte.2017.08.002.

[KS20] F. Kuhn and P. Schneider. "Computing Shortest Paths and Diameter in the Hybrid Network Model". In: *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC).* 2020, pp. 109–118. DOI: 10.1145/3382734.3405719.

[Le 16] F. Le Gall. “Further Algebraic Algorithms in the Congested Clique Model and Applications to Graph-Theoretic Problems”. In: *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*. 2016, pp. 57–70. DOI: 10.1007/978-3-662-53426-7_5.

[Lei+94] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. “Randomized Routing and Sorting in Fixed-Connection Networks”. In: *Journal of Algorithms* 17.1 (1994), pp. 157–205. DOI: 10.1006/jagm.1994.1030.

[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)*. 2013, pp. 42–50. DOI: 10.1145/2484239.2501983.

[Li20] J. Li. “Faster Parallel Algorithm for Approximate Shortest Path”. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. 2020, pp. 308–321. DOI: 10.1145/3357713.3384268.

[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.

[LPP06] Z. Lotker, B. Patt-Shamir, and D. Peleg. “Distributed MST for constant diameter graphs”. In: *Distributed Computing* 18.6 (2006), pp. 453–460. DOI: 10.1145/383962.383984.

[Lub86] M. Luby. “A Simple Parallel Algorithm for the Maximal Independent Set Problem”. In: *SIAM Journal on Computing* 15.4 (1986), pp. 1036–1053. DOI: 10.1137/0215074.

[Lun+10] K. Lund, A. J. Manzo, N. Dabby, N. Michelotti, A. Johnson-Buck, J. Nangreave, S. Taylor, R. Pei, M. N. Stojanovic, N. G. Walter, E. Winfree, and H. Yan. “Molecular robots guided by prescriptive landscapes”. In: *Nature* 465.7295 (2010), pp. 206–210. DOI: 10.1038/nature09012.

[LW00] Y.-F. Lan and Y.-L. Wang. “An optimal algorithm for solving the 1-median problem on weighted 4-cactus graphs”. In: *European Journal of Operational Research* 122.3 (2000), pp. 602–610. DOI: 10.1016/S0377-2217(99)00080-6.

[LWS99] Y.-F. Lan, Y.-L. Wang, and H. Suzuki. “A linear-time algorithm for solving the center problem on weighted cactus graphs”. In: *Information Processing Letters* 71.5-6 (1999), pp. 205–212. DOI: 10.1016/S0020-0190(99)00111-8.

[Mar12] E. Markou. “Identifying Hostile Nodes in Networks Using Mobile Agents”. In: *Bulletin of the European Association for Theoretical Computer Science* 108 (2012), pp. 93–129.

[Mét+11] Y. Métivier, J. M. Robson, N. Saheb-Djahromi, and A. Zemmari. “An optimal bit complexity randomized distributed MIS algorithm”. In: *Distributed Computing* 23.5-6 (2011), pp. 331–340. DOI: `10.1007/s00446-010-0121-5`.

[Min67] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., 1967. ISBN: 0-13-165563-9.

[MKK94] S. Murata, H. Kurokawa, and S. Kokaji. “Self-Assembling Machine”. In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation (ICRA).* 1994, pp. 441–448. DOI: `10.1109/ROBOT.1994.351257`.

[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`.

[Nag52] J. Nagura. “On the interval containing at least one prime number”. In: *Proceedings of the Japan Academy* 28.4 (1952), pp. 177–181. DOI: `10.3792/pja/1195570997`.

[Nan14a] D. Nanongkai. “Distributed Approximation Algorithms for Weighted Shortest Paths”. In: *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC).* 2014, pp. 565–573. DOI: `10.1145/2591796.2591850`.

[Nan14b] D. Nanongkai. *Distributed Approximation Algorithms for Weighted Shortest Paths.* 2014. arXiv: `1403.5171 [cs.DS]`.

[Nas64] C. S. J. A. Nash-Williams. “Decomposition of Finite Graphs Into Forests”. In: *Journal of the London Mathematical Society* s1-39.1 (1964), pp. 12–12. DOI: `10.1112/jlms/s1-39.1.12`.

[Nie+20] E. Niehs, A. Schmidt, C. Scheffer, D. E. Biediger, M. Yannuzzi, B. Jenett, A. Abdel-Rahman, K. C. Cheung, A. T. Becker, and S. P. Fekete. “Recognition and Reconfiguration of Lattice-Based Cellular Structures by Simple Robots”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA).* 2020, pp. 8252–8259. DOI: `10.1109/ICRA40945.2020.9196700`.

[NMN01] J. Nešetřil, E. Milková, and H. Nešetřilová. “Otakar Borůvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history”. In: *Discrete Mathematics* 233.1-3 (2001), pp. 3–36. DOI: `10.1016/S0012-365X(00)00224-7`.

[Now18] K. Nowicki. *Random Sampling Applied to the MST Problem in the Node Congested Clique Model.* 2018. arXiv: `1807.08738 [cs.DS]`.

[ORS07] M. Onus, A. Richa, and C. Scheideler. “Linearization: Locally Self-stabilizing Sorting in Graphs”. In: *Proceedings of the 9th Workshop on Algorithm Engineering & Experiments (ALENEX).* 2007, pp. 99–108. DOI: `10.1137/1.9781611972870.10`.

[OSS09] T. Omabegho, R. Sha, and N. C. Seeman. “A Bipedal DNA Brownian Motor with Coordinated Legs”. In: *Science* 324.5923 (2009), pp. 67–71. DOI: 10.1126/science.1170336.

[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.

[Pel00] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000. ISBN: 978-0-89871-464-7. DOI: 10.1137/1.9780898719772.

[Pel12] A. Pelc. “Deterministic Rendezvous in Networks: A Comprehensive Survey”. In: *Networks* 59.3 (2012), pp. 331–347. DOI: 10.1002/net.21453.

[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).* 1999, pp. 253–261. DOI: 10.1109/SFFCS.1999.814597.

[Pri57] R. C. Prim. “Shortest Connection Networks And Some Generalizations”. In: *The Bell System Technical Journal* 36.6 (1957), pp. 1389–1401. DOI: 10.1002/j.1538-7305.1957.tb01515.x.

[PRT12] D. Peleg, L. Roditty, and E. Tal. “Distributed Algorithms for Network Diameter and Girth”. In: *Proceedings of the 39th International on Colloquium Automata, Languages, and Programming (ICALP).* 2012, pp. 660–672. DOI: 10.1007/978-3-642-31585-5_58.

[Ran91] A. G. Ranade. “How to Emulate Shared Memory”. In: *Journal of Computer and System Sciences* 42.3 (1991), pp. 307–326. DOI: 10.1016/0022-0000(91)90005-P.

[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).* 2001, pp. 329–350. DOI: 10.1007/3-540-45518-3_18.

[Rob21] P. Robinson. “Being Fast Means Being Chatty: The Local Information Cost of Graph Spanners”. In: *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA).* 2021, pp. 2105–2120. DOI: 10.1137/1.9781611976465.126.

[RS09] J. H. Reif and S. Sahu. “Autonomous programmable DNA nanorobotic devices using DNAzymes”. In: *Theoretical Computer Science* 410 (2009), pp. 1428–1439. DOI: 10.1016/j.tcs.2008.12.003.

[RS11] M. Rossberg and G. Schaefer. “A Survey on Automatic Configuration of Virtual Private Networks”. In: *Computer Networks* 55.8 (2011), pp. 1684–1699. DOI: 10.1016/j.comnet.2011.01.003.

[RW00] P. W. K. Rothemund and E. Winfree. “The Program-Size Complexity of Self-Assembled Squares”. In: *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*. 2000, pp. 459–468. DOI: 10.1145/335305.335358.

[Sch98] C. Scheideler. *Universal Routing Strategies for Interconnection Networks*. Springer Verlag, Heidelberg, 1998. ISBN: 978-3-540-64505-4. DOI: 10.1007/BFb0052928.

[Sha74] A. N. Shah. “Pebble Automata on Arrays”. In: *Computer Graphics and Image Processing* 3.3 (1974), pp. 236–246. DOI: 10.1016/0146-664X(74)90017-3.

[SM15] D. Soldani and A. Manzalini. “Horizon 2020 and Beyond: On the 5G Operating System for a True Digital Society”. In: *IEEE Vehicular Technology Magazine* 10.1 (2015), pp. 32–42. DOI: 10.1109/MVT.2014.2380581.

[SP04] J.-S. Shin and N. A. Pierce. “A Synthetic DNA Walker for Molecular Transport”. In: *Journal of the American Chemical Society* 126.35 (2004), pp. 10834–10835. DOI: 10.1021/ja047543j.

[SSS95] J. P. Schmidt, A. Siegel, and A. Srinivasan. “Chernoff-Hoeffding Bounds for Applications with Limited Independence”. In: *SIAM Journal on Discrete Mathematics* 8.2 (1995), pp. 223–250. DOI: 10.1137/S089548019223872X.

[Sto+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. “Chord: A scalable peer-to-peer lookup service for internet applications”. In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160. DOI: 10.1145/964723.383071.

[Tel+18] A. Tell, W. Babalola, G. Kalebiala, and K. Chinta. “SD-WAN: A Modern Hybrid-WAN to Enable Digital Transformation for Businesses”. In: *IDC White Paper* (Apr. 2018).

[Thu+17] A. J. Thubagere, W. Li, R. F. Johnson, Z. Chen, S. Doroudi, Y. L. Lee, G. Izatt, S. Wittman, N. Srinivas, D. Woods, E. Winfree, and L. Qian. “A cargo-sorting DNA robot”. In: *Science* 357.6356 (2017). DOI: 10.1126/science.aan6558.

[TM08] Y. Terada and S. Murata. “Automatic Modular Assembly System and its Distributed Control”. In: *International Journal of Robotics Research* 27.3–4 (2008), pp. 445–462. DOI: 10.1177/0278364907085562.

[Tom+99] K. Tomita, S. Murata, H. Kurokawa, E. Yoshida, and S. Kokaji. “Self-Assembly and Self-Repair Method for a Distributed Mechanical System”. In: *IEEE Transactions on Robotics and Automation* 15.6 (1999), pp. 1035–1045. DOI: 10.1109/70.817668.

[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.

[Upf84] E. Upfal. "Efficient Schemes for Parallel Communication". In: *Journal of the ACM (JACM)* 31.3 (1984), pp. 507–517. DOI: 10.1145/828.1892.

[UY91] J. D. Ullman and M. Yannakakis. "High-Probability Parallel Transitive-Closure Algorithms". In: *SIAM Journal on Computing* 20.1 (1991), pp. 100–125. DOI: 10.1137/0220006.

[Wan+10] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan. "c-Through: Part-time Optics in Data Centers". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. 2010, pp. 327–338. DOI: 10.1145/1851182.1851222.

[WEW12] Z.-G. Wang, J. Elbaz, and I. Willner. "A Dynamically Programmed DNA Transporter". In: *Angewandte Chemie International Edition* 51.18 (2012), pp. 4322–4326. DOI: 10.1002/anie.201107855.

[Wic+12] S. F. Wickham, J. Bath, Y. Katsuda, M. Endo, K. Hidaka, H. Sugiyama, and A. J. Turberfield. "A DNA-based molecular motor that can navigate a network of tracks". In: *Nature Nanotechnology* 7.3 (2012), pp. 169–173. DOI: 10.1038/nnano.2011.253.

[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)*. 2013, pp. 353–354. DOI: 10.1145/2422436.2422476.

[Zho+12] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. "Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers". In: *Proceedings of the ACM SIGCOMM 2012 Conference*. 2012, pp. 443–454. DOI: 10.1145/2342356.2342440.