Construction, Reconfiguration and Computation in Actively Dynamic Networks

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy by

George Skretas

November 2022

Contents

Illustrations

List of Figures

List of Tables

Abstract

Networks can be found in our every day life and they can range from physical transportation networks to virtual social networks. A key characteristic of such networks is their dynamicity, in other words, the changes in their structure. The algorithmic theory of dynamic networks is tasked with investigating such networks and devise algorithms that are tailor-made for these highly dynamic settings.

In this thesis, we are interested in studying *actively* dynamic networks, which are networks where the changes in the structure are controlled by the algorithm that runs on the network. In doing so, we introduce new models and investigate existing frameworks for both centralized and distributed cases. Our goal is to explore the capabilities of such networks and find out which tasks can be solved in these settings. When possible, we provide algorithms that solve these tasks.

In Chapter 2, we introduce a new model where the entities of the network can activate new connections with other entities that are "close" or deactivate pre-existing connections. Our goal is to exploit the ability of the entities to activate new connections in order to efficiently solve some distributed tasks, such as leader election. We show that there is a very simple procedure that achieves this but it requires a large amount of edge activations which is an unrealistic assumption for standard networks. To quantify this cost of activating edges, we introduce different edge complexity measures and provide efficient algorithms that reconfigure the networks in order to minimize the running time required to solve the distributed task while also minimizing the cost introduced by the edge complexity measures.

In Chapter 3, we also introduce a new model where the entities of the network have the unique ability to generate new entities. Once generated, new entities can activate new connections based on their distance from other entities. We investigate the task of constructing a target network G starting from a single entity and we provide centralized algorithms for constructing different classes of networks. We note that apart from the question of whether a network G can be constructed at all, there seems to be a trade off between how fast one can construct network G and how many excess connections need to be activated.

In Chapter 4, we investigate an existing framework where the entities of the network are placed on a 2-dimensional grid. We study whether an initial *shape* A can *transform* to some target shape B by a sequence of movements, called *rotation* and *sliding*. In the beginning, we only focus on the case where rotation is available to the entities and afterwards we study the case in which both rotation and sliding are available. In both cases, we provide feasibility results.

Acknowledgements

I am very lucky to have Othon as my supervisor. He was the first person to inspire me to do research and he has guided me so far throughout my whole career and I could not be more grateful for that. Had it not been for you, I do not think that I would be doing so well in my career and life. I hope that I can also do my part and transmit the knowledge and passion that you gave to me to other young researchers as well. I would also like to thank Paul for always pushing me to do more and helping me when I struggled, and Leszek for guiding me through my first years as a PhD student.

Many thanks to my academic advisors Igor Potapov and Kostantinos Tsakalidis for assessing my progress every year and giving me advice on how to improve my presentation skills. I would also like to thank all of the people I had the honor of working with: Othon, Paul, Michael Theofilatos, Argyrios Deligkas, Edward Eiben and George Mertzios. I learned a lot of things from them which helped me grow as a person and researcher.

I would also like to thank the people from the Networks and Distributed Computing group for showing me how academia works and introducing to many different topics. Furthermore, I would like to thank the University of Liverpool for the financial support that made my Ph.D. studies possible and gave me the privilege to meet and work with all these great people.

Finally, I would like to mention all the people that went through this period with me. I did not think that moving to a new country could ever be easy to do but they actually made it an unbelievably wonderful experience for me. Thank you Michali, Katerina, Argy, Katerina, Ari, Elektra, Eleni, Themi, Mano, Nico, Eleni, Fonta, Matina, Alkmini, Dimitri, Katerina, Tobenna, Tonia, Valia. I would also like to thank my friends in Greece, and most importantly, my family: Marella, Stathis and Fiorella for encouraging me to get out there and succeed in what I enjoy. I am not sure I could have done this without them.

Chapter 1

Introduction

1.1 Actively Dynamic Networks

The algorithmic theory of dynamic networks is a research area in the field of theoretical computer science tasked with investigating the algorithmic and structural properties of networked systems whose structure changes over time.

This area has risen in popularity which is partly due to the multitude of motivating systems that exist for such frameworks. One example includes wireless systems such as peer-to-peer or ad hoc networks. Due to the low cost wireless systems that currently exist and the limitations of communication via physical means such as cables, there is an increasing demand for wireless communication systems. Such networks are inherently dynamic and their structure is ever changing. Dynamic networks can be classified into different categories based on multiple properties. One of the most important ones, and one that is of vital importance to the present thesis, is based on *who controls the dynamics*. In *passively* dynamic networks the changes are external to the algorithm, in the sense that the algorithm has no control over them. Such dynamics are usually modeled by sequences of events determined by an *adversarial scheduler*. Therefore, the algorithm running on the network must account for the changes that are out of its control. There are many models that try to encapsulate the properties of different passively dynamic networks that exist. One of the earliest bodies of work [1, 2], assumed that an initial dynamic graph would, at some point, stabilize and changes in the structure would halt. Other models include the population protocol model introduced by Angluin *et al.* [3] where the structure of the network is fully dynamic and only one connection between the entities of the network is activated by an adversarial scheduler per round, and the model introduced by Kuhn *et al.* [4], where multiple connections can be active per round but again there is an adversarial scheduler that controls them.

In other applications, the entities can *actively* control the dynamics of their network. Many examples are apparent, such as social networks where the user decides the

connections she/he can have with other users, wireless communication networks [5] and peer-to-peer systems [6] where the software decides the connections between endpoints in order to optimize some useful properties needed, transportation networks [7] where new routes can be introduced in the system or old ones can be discarded and reconfigurable robotics [8, 9] where the robots move around to change the overall structure. We must also point out that actively dynamic networks are, in general more powerful than static ones. This is due to the fact that actively dynamic networks can reconfigure in a more optimal structure depending on the task at hand that they are trying to accomplish. Therefore, if we consider a standard problem in static networks, such as leader election, the same problem in an actively dynamic network is quite different and we must devise new algorithms in order to take advantage of the new potential that is introduced. There is also a surge of bio-inspired systems. Driven by the realization of the fact that many natural processes are essentially algorithmic and by the advent of some low-cost and accessible technologies for programming and controlling various forms of matter, there is a growing interest in abstracting biological processes and formalizing their algorithmic principles [10–12] and in inventing new algorithmic techniques suitable for the existing technologies. The above fact combined with the ongoing effort to set the algorithmic foundations of dynamic networks has urged scientists to develop models that try to mimic biological systems.

We can surmise that actively dynamic networks are evident in real-life systems and we should strive to increase our understanding of their capabilities and limitations. It is also important to devise new algorithms that are tailor-made for such systems since it also seems that previous algorithms that were implemented for passively dynamic networks are very inefficient. Passively dynamic networks have to account for the changing structure of the system for which the algorithm running on the network has no control over, whereas in actively dynamic networks, the algorithm has complete control over the network and solving tasks can become more efficient.

Dynamic networks are usually modelled using graph theory where the network is represented by a graph and the nodes represent the entities of the network and the edges represent the connections between the entities. The dynamicity of the network usually arises from two changes: (i) new entities may appear in the system or existing entities may disappear and (ii) pre-existing connections may become unavailable or new connections might be established. Every model provided in this thesis can be described as a dynamic/temporal graph that changes over time.

A static graph G is a pair (V, E), where V is a finite set of *vertices* or *nodes* (usually denoted u or v), and E is a set of *edges*, each being an unordered pair uw of distinct nodes. In a dynamic graph G, either set V, or E, or both change with *time/rounds*. A snapshot of a dynamic graph G at round t, denoted G_t, can be described as $G_t = (V(t), E(t))$ where set $V(t)$ contains the nodes and $E(t)$ contains the edges

that are *active* at round t. We define the *lifetime* of a dynamic graph to be equal to the number of rounds for which the dynamic graph exists. Therefore, if we consider r to be the lifetime of a graph , the dynamic graph G can be defined as the union of all graphs G_t, $G = (G_1 \cup G_2, \cup \ldots, \cup G_t)$, for $t = 1, 2, \ldots, r$. The *size* of the graph (number of distinct nodes) is denoted as n. Since each of the chapters studies a different model, we will give a more detailed description at the beginning of each chapter.

An important classification for such models comes from *who controls the algorithm*. This gives rise to two main families of models. One is *fully centralized*, in which a central controller has global view of the system. In case of active network dynamics, the centralized algorithm typically reconfigures a dynamic network by exploiting its full knowledge about the system in a way that aims to optimize some given objective function. Similar objectives hold for the *fully distributed* case, in which every node in the network is an independent computing entity, like an automaton or Turing machine, typically equipped with computation and communication capabilities, and in the case of active dynamics, with the additional capability to locally modify the network structure. Based on the motivating factors previously mentioned in this chapter, the distributed case is closer to the real-life applications but the centralized model also plays an important role for two reason: (i) as a precursor to the distributed case, since the centralized model can give unique insight the intricacies of the distributed case and, (ii) as its own application since there are many real-life systems where the control is centralized, such as transportation networks. We are going to explore both settings.

1.2 Roadmap

In this thesis, we study different models of actively dynamic networks. Our goal is to study real-life systems that are not yet very well explored in the literature. In doing so, we investigate existing frameworks or introduce new models and we study their capabilities and limitations depending on the constraints imposed by each model. In Chapter 2, we introduce a new model with dynamic connections and distributed control where the nodes of the network can manipulate their connections. The nodes are able to activate new connections with their neighbors or deactivate part of their current connections. The goal is to reconfigure any initial network into a target network with small diameter under some constraints based on the number of connections activated throughout the execution of the algorithm. In Chapter 3, we introduce a model where nodes are equipped with the ability to generate (give birth) new nodes in the network and newly generated nodes can activate new connections with their neighbors at the time of their birth. The goal is to generate a target network, starting from the *singleton* network G_0 that contains a single node. In Chapter 4, we investigate a variation of an existing programmable matter model [13] where nodes reside on a *2D lattice grid*.

The nodes are allowed to execute one or both of two following movements: (i) rotation and (ii) sliding. Given the above movement restriction, we investigate reconfiguration problems pertaining to transforming any initial network A to a target networks B.

In the following sections of the Introduction we give a brief overview of models and the results of each part of the thesis. The detailed analysis, related work and results for each model are presented in the corresponding chapter.

1.3 Distributed Computation and Reconfiguration

In Chapter 2, we consider an *actively dynamic* and fully distributed system where we have a static set of entities and a dynamic set of connections. The motivation of this model stems from wireless networks. We want to introduce a model for wireless communication where the connections between the entities are limited by their distance but the algorithm is able to manipulate the connections. By reconfiguring the network, we are able to develop a "better" structure for the network that allows the entities to coordinate in a timely manner to compute otherwise difficult or impossible tasks.

In the model introduced in chapter 2, there are n computing entities starting from an *initial connected network* drawn from a family of initial networks. The entities are typically equipped with unique IDs, can compute locally, can communicate with neighboring entities, and can activate connections to new neighbors locally or eliminate some of their existing connections. All these take place in lock step through a standard synchronous message passing model, extended to include the additional operations of edge activations and deactivations within each round. The goal is, generally speaking, to program all the entities with a distributed algorithm that can transform the initial network G_s into a *target network* G_f with small diameter. A small diameter would guarantee that the network is compact and coordination can happen very efficiently due to the small distance between any two nodes. As such, disseminating information throughout the network can happen very fast. Note that there is a trivial process that can transform any initial network into a *clique* graph. But we mentioned that our model is inspired from systems that have limitations on their structure, it is naive to think that a clique network would ever be possible in a real system. Therefore, we try to capture the cost incurred by activating and maintaining connections, by adding edge complexity measures related to the edges and the degree of each node.

1.4 Growing Graphs

In the past century, and quite naturally, computing research has primarily concerned itself with understanding the fundamental principles of numerical computation. This

has led to the development of a beautiful theory of computation and its various subtheories including complexity theory, distributed computing, algorithmic theories of graphs, learning, and games. Driven by the realization of the fact that many natural processes are essentially algorithmic and by the advent of some low-cost and accessible technologies for programming and controlling various forms of matter, there is a growing interest in abstracting biological processes and formalizing their algorithmic principles and in inventing new algorithmic techniques suitable for the existing technologies. Biological systems have a unique ability to multiply into large structures starting from single cells through the process called embryogenesis. The information for this process exists in the genetic code of each cell and it contains the construction program for this complex cellular division that includes controlled tissue growth. Motivated by this progress and by the algorithmic principles of biological development, our goal here is to study an abstraction of networked systems which, starting from a single entity, can grow into well-defined global networks and structures.

In chapter 3, a *growing graph* is modeled as an undirected dynamic graph $G_t = (V_t, E_t)$ for $t = 1, 2, \ldots, k$. The initial graph G_0 consists only of a single node (singleton), called the *initiator*. During the execution, the graph is gradually changing through the addition of nodes and edges and/or deletion of edges. Every node u (parent) has the capability of generating a new node v (child). Each newly generated child w is initially connected (by default) to its parent, and can additionally activate some edges, only at the the time of its birth t, with any node v such that the distance between w and v, at the time of the birth, is at most d. We call d the *edge-activation distance*. We only allow nodes to activate edges at the time of their birth because we want to model biological systems, where entities cannot receive large amounts of information to establish new connections but each entity can receive the information its parent has (via replication at the moment of its birth). If we assumed that every entity can establish new connections with other entities in close proximity, allowing edge activations at any point would mean that the network is somehow becoming more compact in size as time goes on. Finally, every node u is able to deactivate any incident edge.

The problem that we examine is the construction of a given connected graph G, starting from G_0 using the operations listed. An interesting attempt in this direction was the Nubot model of Woods *et al.* [14], showing how to efficiently self-assemble shapes and patterns from simple monomers in a 2D discrete geometric environment. Apart from the Nubot model the literature seems lacking on the subject of growing networks. While the ultimate goal is to study the distributed case of the above model, it seems prudent to begin with the centralized case in order to find the limitations of the model as well as key intuitions that might translate from the centralized case to the distributed one.

1.5 Programmable Matter

While Chapters 2 and 3 focused on abstract dynamic networks, we now want to study models with geometrical properties. One such field that studies geometrical and actively dynamic networks is the programmable matter area.

Programmable matter refers to any type of matter that can *algorithmically* change its physical properties. "Algorithmically" means that the change (or *transformation*) is the result of executing an *underlying program*. Depending on the implementation, the program could either be a *centralized algorithm* capable of controlling the whole programmable matter system (*external* control) or a *decentralized protocol* stored in the material itself and executed by various sub-components of the system (*internal* control). For a concrete example, imagine a material formed by a collection of spherical (nano- or micro-) modules kept together by magnetic or electrostatic forces. Each module is capable of storing (in some internal representation) and executing a simple program that handles communication with nearby modules and that controls the module's electromagnets/capacitors, in a way that allows the module to *rotate* or *slide* over neighboring modules. Such a material would be able to adjust its *shape* in a programmable way. Other examples of physical properties of interest for real applications would be connectivity, color [15, 16], and strength of the material.

Computer scientists, nanoscientists, and engineers are more and more joining their forces towards the development of such programmable materials and have already produced some first impressive outcomes (even though it is evident that there is much more work to be done in the direction of real systems), such as programmed DNA molecules that self-assemble into desired structures [11, 17] and large collectives of tiny identical robots that orchestrate resembling a single multi-robot organism (e.g., the Kilobot system [12]). Other systems for programmable matter include the Robot Pebbles [18], consisting of 1cm cubic programmable matter modules able to form 2-dimensional (abbreviated "2D" throughout) shapes through self-disassembly, and the Millimotein [19], a chain of programmable matter which can fold itself into digitized approximations of arbitrary 3-dimensional (abbreviated "3D" throughout) shapes. Ambitious long-term applications of programmable materials include molecular computers, collectives of nanorobots injected into the human circulatory system for monitoring and treating diseases, or even self-reproducing and self-healing machines.

In chapter 4, we follow such an approach, by studying the transformation capabilities of models for programmable matter, which are based on minimal mechanical capabilities, easily implementable by existing technology. We study a minimal programmable matter system consisting of n cycle-shaped modules, with each module (or *node*) occupying at any given time a cell of the 2D grid (no two nodes can occupy the same cell at the same time). Therefore, the composition of the programmable matter

systems under consideration is discrete. Our main question throughout is whether an initial arrangement of the material can *transform* (either in principle, e.g., by an external authority, or by itself) to some other target arrangement. Usually, a step consists either of a *valid movement* of a single node (in the *sequential case*) or of more than one node at the same time (in the *parallel case*). We consider two quite primitive types of movement. The first one, called *rotation*, allows a node to rotate 90 degrees around one of its neighbors either clockwise or counterclockwise and the second one, called *sliding*, allows a node to slide by one position "over" two neighboring nodes.

1.6 Related Work

Distributed Computation in Passively Dynamic Networks. Probably the first authors to consider distributed computation in passively dynamic networks were Angluin *et al.* [3, 20, 21]. Their population protocol model, considered originally the computational power of a population of n finite automata which interact in pairs passively either under an eventual fairness condition or under a uniform random scheduling assumption. A variant of population protocols in which the automata can additionally create or destroy connections between them was introduced in [22, 23]. It was shown that in that model, called network constructors, complex spanning networks can be created efficiently despite the computational weakness of individual entities. Other papers [4, 24, 25] have studied distributed computation in worst-case dynamic networks using a traditional message-passing model and typically operating through local broadcast in the current neighborhood. The key difference between these models and the work in this thesis is that the previous models are passively dynamic networks whereas the models introduced here are exclusively actively dynamic networks. Another model [26] was also introduced where node deletions and insertions are handled by an adversary and the algorithm has to respond accordingly in a determistic manner.

Temporal Graphs. The algorithmic study of temporal graphs was initiated by Berman [27] and Kempe *et al.* [7], who studied a special case of temporal graphs in which every edge can be available at most once and continuing with subsequent work on the multi-labeled case [28–30]. The problem of designing a cost-efficient temporal graph satisfying some given connectivity properties was introduced in [28]. The design task was carried out by an offline centralized algorithm starting from an empty edge set. Subsequent work [29, 31], motivated by epidemiology applications, considered the centralized algorithmic problem of re-designing a given temporal graph through edge deletions in order to end up with a temporal graph with bounded temporal reachability, thus keeping the spread of a disease to a minimum. Our work is related to the temporal network (re-)design problem but the models studied in this thesis are generally more dynamic (nodes are static in the temporal graphs model) and also distributed in cases.

Programmable Matter and Self-Assembly. There is a growing interest in studying the algorithmic foundations of systems that can change their physical properties through local reconfigurations [10, 32–35]. Programmable matter refers to any type of matter that can be programmed to change its physical properties, such as its shape [36]. Progress in small-scale engineering has enabled the production of tiny monads, whose size ranges from milli down to nano and which are equipped with computation, communication, sensing, and actuation capabilities. These monads are bound to neighboring monads, usually by electromagnetic or electrostatic forces, forming a connected 2D or 3D shape and may be programmed to reconfigure in order to adapt to their environment or to solve a task requiring modification of their joint physical properties. Typical examples of systems in this area are reconfigurable robotics, swarm robotics, and self-assembly systems [8, 9, 37]. In most of these settings, modification of structure can be represented as a dynamic network, usually called *shape*, with additional geometric restrictions coming from the shape and the local reconfiguration mechanism of the entities. The goal is to transform a given initial shape into a desired target shape through a sequence of valid local moves. Equally impressive progress has been made in the domain of DNA self-assembly. There, DNA strands have been successfully programmed to self-assemble into any desired nano-scale pattern [17], to implement registers or to simulate circuits [38] and the goal of universally programming molecules seems now to be within reach. Both directions are based on and further inspiring the development of solid algorithmic foundations [11, 39–41].

1.7 Thesis Contribution

1.7.1 Distributed Computation and Reconfiguration in Actively Dynamic Networks

As previously stated, the goal of chapter 2 is to provide distributed algorithms that can reconfigure the initial network G_s into a *target network* G_f with small diameter via edge activations and deletions. To formally capture costs associated with activating and maintaining edges, we define three measures: the *total edge activations*, the *maximum activated edges per round*, and the *maximum activated degree of a node*. We give (poly)$\log(n)$ time algorithms for the task of transforming any G_s into a G_f of diameter (poly)$\log(n)$, while minimizing the edge-complexity. Our main lower bound shows that $\Omega(n)$ total edge activations and $\Omega(n/\log n)$ activations per round must be paid by any algorithm (even centralized) that achieves an optimum of $\Theta(\log n)$ rounds. We give three distributed algorithms for our general task. The first runs in $O(\log n)$ time, with at most $2n$ active edges per round, a total of $O(n \log n)$ edge activations, a maximum degree $n-1$, and a target network of diameter 2. The second achieves bounded degree

by paying an additional logarithmic factor in time and in total edge activations. It gives a target network of diameter $O(\log n)$ and uses $O(n)$ active edges per round. Our third algorithm shows that if we slightly increase the maximum degree to polylog(n) then we can achieve $o(\log^2 n)$ running time.

1.7.2 The Complexity of Growing a Graph

The main problem studied in chapter 3 is given a graph $G = (V, E)$, construct graph G starting from the singleton graph G_0. The feasibility, speed and efficiency of constructing graph G depends on the edge activation distance d allowed by the model. Based on our observations, there seems to be a clear trade-off between the number of rounds required to construct the graph and the excess edges generated by the construction that do not belong to the input graph G. Thus we formally formulate the problem as follows: Given an input graph $G = (V, E)$, compute in polynomial time a construction schedule with k slots (rounds) and l excess edges, if it exists. We show that for $d = 1$, the only graphs that can be constructed are tree graphs and we also provide a time-optimal construction schedule for such graphs. For $d \leq 4$, we provide an algorithm that computes a construction schedule for any graph G with $\lceil \log n \rceil$ slots and $O(n)$ excess edges.

The case where $d = 2$ seems to be the most interesting one. First, we provide lower bounds based on the chromatic number and largest clique size of a graph. We continue by creating simple algorithms for line and star graphs. We then use these algorithms as subroutines and provide an algorithm for tree graphs that computes a construction schedule with $O(\log^2 n)$ slots and $O(n)$ edges, and an algorithm for planar graphs that computes a construction schedule with $O(\log n)$ slots and $O(n \log n)$ edges. We then show that the optimal number of time slots to construct an input target graph with zero-waste (i.e., no edge deletions allowed), is hard even to approximate within $n^{1-\varepsilon}$, for any $\varepsilon > 0$, unless P=NP. On the contrary, the question of the feasibility of constructing a given target graph in $\log n$ time slots and zero-waste, can be answered in polynomial time.

1.7.3 On the Transformation Capability of Feasible Mechanisms for Programmable Matter

In chapter 4, we study whether an initial *shape* A can *transform* to some target shape B by a sequence of movements. The first part of the chapter focuses on the case in which only rotation is available to the nodes and the other part studies the case in which both rotation and sliding are available. The latter case has been studied to some extent in the past in the, so called, *metamorphic systems* [13, 42, 43], which makes those studies the closest to our approach.

When only rotation is available, we introduce the notion of *color-consistency*, and prove that if two shapes are not color-consistent then they cannot be transformed to each other. We also prove that deciding whether two given shapes can transform to each other, can be achieved in polynomial time. Under the additional restriction of maintaining global connectivity, we prove inclusion in PSPACE and explore minimum *seeds* that can make otherwise infeasible transformations feasible. Seeds are external nodes introduced into the system that are not part of the initial shape. Allowing both rotations and slidings yields universality: any two connected shapes of the same order can be transformed to each other without breaking connectivity, in $O(n^2)$ sequential and $O(n)$ parallel time (both optimal). Finally, we assume that the nodes are distributed processes equipped with limited local memory and able to perform communicate-compute-move rounds (where, again, both rotation and sliding movements are available) and provide distributed algorithms for a general type of transformation.

1.8 Author's Publications and Pre-prints Presented in this Thesis

Publications				
Chapter	Title	Authors	Conference	Journal
2	On the Transformation Capability of Feasible Mechanisms for Programmable Matter	Othon Michail, George Skretas, Paul G. Spirakis	ICALP (2017) [1][44]	Journal of Computer and System Sciences (2019) [33]
3	Distributed Computation and Reconfiguration in Actively Dynamic Networks		PODC (2020) [2][45]	Distributed Computing (2021)
4	The Complexity of Growing a Graph	George Mertzios, Othon Michail, George Skretas, Paul G. Spirakis, Michail Theofilatos	ALGOSENSORS 2022 [3][46]	

TABLE 1.1: List of author's publications and pre-prints presented in this thesis

[2]44th International Colloquium on Automata, Languages and Programming
[3]PODC '20: Proceedings of the 39th Symposium on Principles of Distributed Computing
[4]Arxiv version

Chapter 2

Distributed Computation and Reconfiguration

2.1 Introduction

In this chapter, we consider an *actively dynamic and fully distributed* system. In particular, there are n computing entities starting from an *initial connected network* drawn from a family of initial networks. The entities are typically equipped with unique IDs, can compute locally, can communicate with neighboring entities, and can activate connections to new neighbors locally or eliminate some of their existing connections. All these take place in lock step through a standard synchronous message passing model, extended to include the additional operations of edge activations and de-activations within each round.

The goal is, generally speaking, to program all the entities with a distributed algorithm that can transform the initial network G_s into a *target network* G_f from a family of target networks. The idea is that starting from a G_s not necessarily having a good property, like small diameter, the algorithm will be able to "efficiently" reach a G_f satisfying the property. This gives rise to two main objectives, which in some cases might be possible to satisfy at the same time. One is to transform a given G_s into a desired target G_f and the other is to exploit some good properties of G_f in order to more efficiently solve a distributed task, like computation of a global function through information dissemination.

Even when edge activations are extremely local, meaning that an edge uv can only be activated if there exists a node w such that both uw and wv are already active, there is a straightforward algorithmic strategy that can successfully carry out most of the above tasks. In every round, all nodes activate all of their possible new connections, which corresponds to each node u connecting with all nodes v_i that were at distance 2 from u in the beginning of the current round. By a simple induction, it can be shown

that in any round r the neighborhood of every node has size at least 2^r, which implies that a clique K_n is formed in $O(\log n)$ rounds. Such a clique can then be used for global computations, like electing the maximum UID as a leader, or for transforming into any desired target network G_f through eliminating the edges in $E(K_n) \setminus E(G_f)$. All these can be performed within a single additional round.

Even though sublinear global computation and network-to-network transformations are in principle possible through the *clique formation* strategy described above, this algorithmic strategy still has a number of properties which would make it impractical for real distributed systems. As already highlighted in the literature of dynamic networks, (i.e., [28]), activating and maintaining a connection does not come for free and is associated with a cost that the network designer has to pay for. Even if we uniformly charge 1 for every such active connection, the clique formation incurs a cost of $\Theta(n^2)$ total edge activations in the worst case and always produces instances (e.g., when K_n is formed) with as many as $\Theta(n^2)$ active edges in which all nodes have degree $\Theta(n)$.

Our goal in this chapter is to formally define such cost measures associated with the structure of the dynamic network and to give improved algorithmic strategies that maintain the time-efficiency of clique formation, while substantially improving the edge complexity as defined by those measures. In particular, we aim at minimizing the edge complexity, given the constraint of (poly)logarithmic running time. Observe at this point that without any restriction on the running time, a standard distributed dissemination solely through message passing over the initial network, would solve global computation without the need to activate any edges. However, linear running times are considered insufficient for our purposes (even when the goal is to solve traditional distributed tasks). Moreover, strategies that do not modify the input network cannot be useful for achieving network-to-network transformations.

2.1.1 Contribution

We define three cost measures associated with the edge complexity of our algorithms. One is the *total number of edge activations* that the algorithm performed during its course, the second one is the *maximum number of activated edges in any round* by the algorithm, and the third one is the *maximum activated degree of a node in any round*, where the maximum activated degree of a node is defined only by the edges that have been activated by the algorithm.

Our ultimate goal in this chapter is to give *(poly)logarithmic time* algorithms which, starting from *any* connected network G_s, transform G_s into a G_f of *(poly)logarithmic diameter* and at the same time *elect a unique leader*. Such algorithms can then be composed with any algorithm B that assumes an initial network of (poly)logarithmic diameter and has access to a unique leader and unique ids. In case of a static network algorithm B, this for example yields (poly)logarithmic time information dissemination

and computation of any global function on inputs. In case of an actively dynamic network algorithm B, it gives (poly)logarithmic time transformation into any target network from a given family which depends on restrictions related to the edge complexity.

We restrict our focus on *deterministic* algorithms, that is, the computational entities do not have access to any random choices. Moreover, our algorithms never break the connectivity of the network of active edges as this would result in components that could never be reconnected based on the permissible edge activations. Temporary disconnections within a round may be permitted but can always be avoided by first activating all new edges and then deactivating any edges for the current round.

There is a clear tradeoff between time and edge complexity and formally capture that with the lower bounds presented in Section 2.7. In particular, we first prove that $\Omega(\log n)$ is a lower bound on time following from an upper bound of 2 on the distance of new connections and the $\Theta(n)$ worst-case diameter of the initial network. Then we give an $\Omega(n)$ lower bound on total edge activations and $\Omega(n/\log n)$ activations per round for any centralized algorithm that achieves an optimal $\Theta(\log n)$ time. Our main lower bound is a total of $\Omega(n \log n)$ total edge activations that any logarithmic time deterministic distributed *comparison based* algorithm must pay. This is in contrast to the $\Theta(n)$ total edges that would be sufficient for a centralized algorithm and is due to the distributed nature of the systems under consideration.

We then proceed to our main positive results. In particular, we give three algorithms for transforming *any* initial connected network G_s into a network G_f of (poly)logarithmic diameter and at the same time electing a unique leader. Each of these algorithms makes a different contribution to the time vs edge complexity tradeoff. All of our main algorithms are built upon the following general strategy. For each of them, we define a different *gadget network* and the algorithms are developed in such a way that they always satisfy the following invariants. In any round of an execution, the network is the union of committees being such gadget networks of varying sizes and some additional edges including the initial edges and other edges used to join the committees. Initially, every node forms its own committee and the algorithms progressively merge pairs or larger groups of committees based on the rule that the committee with the greater UID dominates. If properly performed, this ensures that eventually only one committee remains, namely, the committee of the node u_{max} with maximum UID in the network. The diameter of all our gadgets is (poly)logarithmic in their size, which facilitates quick merging and ensures that the final committee of u_{max} satisfies the (poly)$\log(n)$ diameter requirement for G_f. The algorithms also ensure that, by the time the committee of u_{max} is the unique remaining committee, u_{max} is the unique leader elected.

Our algorithms must achieve (poly)logarithmic time and they do so by satisfying the invariant that surviving committees always grow exponentially fast. This growth is *asynchronous* in our algorithms for the following reason. In a typical configuration (of a phase) the graph of mergings forms a spanning forest F of committees such that any tree T in F is rooted at the committee that will eventually consume all committees in $V(T)$. Given that those trees may have different sizes (even up to $V(T) = \Theta(n)$), the rounds in which various committees finish merging may be different, but we can still show that their amortized growth is exponential.

Our first algorithm, called `GraphToStar` and presented in Section 2.4, uses a star network as a gadget. Its running time is $O(\log n)$ and it uses at most $2n$ active edges per round and an optimal total of $O(n \log n)$ edge activations. The target network G_f that it outputs is a spanning star, thus, achieving a final diameter of 2.

Our second algorithm, called `GraphToWreath` and presented in Section 2.5, uses as a gadget a graph we call a *wreath* which is the union of a ring and a complete binary tree spanning the ring. The main improvement compared to `GraphToStar` is that it maintains a bounded maximum degree throughout its course (given a bounded-degree G_s). It does this at the cost of increasing the running time to $O(\log^2 n)$ and the number of total edge activations to $O(n \log^2 n)$. The active edges per round remain $O(n)$. The target network G_f that it outputs is a spanning complete binary tree (after deleting the original edges and the spanning ring), thus, the algorithm achieves a final diameter of $O(\log n)$.

Our third algorithm, called `GraphToThinWreath` and presented in Section 2.6, shows that if we slightly increase the maximum degree to polylog(n) then we can achieve a running time of $o(\log^2 n)$ (more precisely, $O(\log^2 n / \log\log^k n)$, for some constant $k \geq 1$).

If our model can be compared to models from the area of overlay networks construction (see Section 2.2 for a discussion on this matter), then `GraphToWreath` is, to the best of our knowledge, the first deterministic bounded-degree $O(\log^2 n)$-time algorithm and `GraphToThinWreath` is the first deterministic polylog(n)-degree $o(\log^2 n)$-time algorithm for the problem of transforming any connected G_s into a polylog(n) diameter G_f.

2.2 Further Related Work

Construction of Overlay Networks. There is a rich literature on the distributed construction of overlay networks. A typical assumption is that there is an overlay (active) edge from a node u to a node v in a given round iff u has obtained v's UID through a message. Without further restrictions, the overlay in round r would always correspond to the union of r consecutive transitive extensions starting from the original

edge set. The main restriction imposed in the relevant literature is a polylogarithmic (in bits) communication capacity per node per round, which also implies that in every round $O(\log n)$ new overlay connections per node are permitted.

Our model and results, even though different in motivation, in the complexity measures considered, and in the restrictions we impose, appear to have similarities with some of the developments in this area. Unlike our work, where our complexity measures are motivated by the cost of creating and maintaining physical or virtual connections, the algorithmic challenges in overlay networks are mainly due to restricting the communication capacity of each node per round to a polylogarithmic total number of bits.

Research in this area started with seminal papers such as Chord of Stoica *et al.* [6] and the Skip graphs of Aspnes and Shah [47]. Probably the first authors to have considered the problem of constructing an overlay network of logarithmic diameter were Angluin *et al.* [5]. Their algorithm is randomized with $O((d + W) \log n)$ running time w.h.p., where W is the maximum size of a unique UID. Then Aspnes and Wu [48] gave a randomized $O(\log n)$ time algorithm for the special case in which the initial network has outdegree 1. A work by Götte *et al.* [49] has improved the upper bound of [5] to $O(\log^{3/2} n)$, w.h.p. It is a randomized algorithm which uses a core deterministic procedure that has some similarities to our algorithmic strategy of maintaining and merging committees (called "supernodes" there) whose size increases exponentially fast. Their model keeps the polylogarithmic restriction on communication and the polylogarithmic maximum degree. Finally, Götte *et al* [50] have proposed a randomized algorithm that runs in $O(\log n)$ optimal time whereas, in different direction, Gilbert *et al* [51] propose another model whose goal is to construct efficient topology

To the best of our knowledge, the only previous deterministic algorithm for the problem is the one by Gmyr *et al.* [52]. Our algorithmic strategies appear to have some similarities to their "Overlay Construction Algorithm", which in their work is used as a subroutine for monitoring properties of a passively dynamic network. Unlike our model, their model is hybrid in the sense that algorithms have partial control over the connections of an otherwise passively dynamic network. Due to using different complexity measures and restrictions it is not totally clear to us yet whether a direct comparison between them would be fair. Still, we give some first observations. Their algorithm has the same time complexity, i.e., $O(\log^2 n)$, with our `GraphToWreath` algorithm, while our `GraphToStar` algorithm achieves $O(\log n)$ and our `GraphToThinWreath` $o(\log^2 n)$. Their overlays appear to maintain $\Theta(n \log n)$ active connections per round, while our algorithms maintain $O(n)$. Their maximum active degree is polylogarithmic, the same as `GraphToThinWreath`, while `GraphToStar` uses linear and `GraphToWreath` always bounded by a constant. Their model restricts the communication capacity of

every node to a polylogarithmic number of bits per round, whereas we do not restrict communication.

Scheideler and Setzer [53] studied the (centralized) computational complexity of computing the optimum graph transformation and gave **NP**-hardness results and a constant-factor approximation algorithm for the problem.

2.3 Preliminaries

2.3.1 Model

An actively dynamic network is modeled in this chapter by a temporal graph $D = (V, E)$, where V is a static set of n nodes and $E \subseteq \binom{V}{2} \times \mathbb{N}$ is a set of undirected time-edges. In particular, $E(i) = \{e : (e, i) \in E\}$ is the set of all edges that are *active* in the temporal graph at the beginning of round i. Since V is static, $E(i)$ can be used to define a snapshot of the temporal graph at round i, which is the static graph $D(i) = (V, E(i))$.

The temporal graph D of an execution is generated by local operations performed by the nodes of the network, starting from an initial graph $G_s = D(1)$. Throughout this chapter, G_s is assumed to be connected. A node u can *activate an edge* with node v in round i (add edge uv in $E(i)$, if $uv \notin E(i)$ and there exists a node w such that both uw and wv are active at the beginning of round i. A node u can *deactivate an edge* with node v in round i (remove edge uv from $E(i)$), provided that $uv \in E(i)$. An active edge remains active indefinitely unless a node that is incident to that edge deactivates it. There is at most one active edge between any pair of nodes, that is multiple edges are not allowed. If a node attempts to activate an edge which is already active, the action has no effect and the edge remains active; similarly for deactivating inactive edges. Moreover, if a node u decides to activate an edge with a node v in round i and v decides to activate an edge with u in the same round, then only one edge is activated between them. In case u and v disagree on their decision about edge uv, then their actions have no effect on uv. We define $E_{ac}(i)$ as the set of all edges that were activated in round i and $E_{dac}(i)$ as the set of all edges that were deactivated in round i. Then $E(i+1) = (E(i) \cup E_{ac}(i)) \setminus E_{dac}(i)$.

We define set $N_1^i(u)$ of node u, where $v \in N_1^i(u)$ iff $uv \in E(i)$ which means that set $N_1^i(u)$ contains the neighbors of node u in round i. Additionally, set $N_2^i(u)$ of node u, where $w \in N_2^i(u)$ iff there exists $v \in V$ s.t. $v \in N_1^i(u)$ and $v \in N_1^i(w)$ and $w \notin N_1^i(u)$. That is, set $N_2^i(u)$ of node u in round i contains the nodes at distance 2 which we will refer to as *potential neighbors*. We will omit the i index for rounds, when clear from context.

Each node $u \in V$ is identical to every other node v but for the unique identifier *(UID)* that each node possesses. Each node u starts with a UID that is drawn from a namespace $\mathcal{U}$. The maximum UID is represented by $O(\log n)$ bits. An algorithm is called *comparison based* if it manipulates the UIDs of the network using comparison operations $(<, >, =)$ only. All of the algorithms and lower bounds presented in this chapter are comparison based.

The nodes represent agents equipped with computation, communication, and edge-modification capabilities and they operate in synchronous rounds. In each round all agents perform the following actions in sequence and in lock step: Send messages to their neighbors, Receive messages from their neighbors, Activate edges with potential neighbors, Deactivate edges with neighbors, Update their local state.

We note that a node may choose to send a different message to different neighbors in a round and that the time needed for internal computations is assumed throughout to be $O(1)$. We do not impose any restriction on the size of the local memory of the agents, still the space complexity of our algorithms is within a reasonable polynomial in n.

2.3.2 Problem Definitions and Performance Measures

In this chapter, we are mainly interested in the following problems.

Leader Election. Every node u in graph $D = (V, E)$ has a variable $status_u$ that can be set to a value in {Follower, Leader}. An algorithm A solves leader election if the algorithm has terminated and exactly one node has its status set to Leader while all other nodes have their status set to Follower.

Token Dissemination. Given an initial graph $D = (V, E)$ where each node $u \in V$ starts with some unique piece of information (token), every node $u \in V$ must terminate while having received that unique piece of information from every other node $v \in V \setminus \{u\}$. W.l.o.g. we will consider that unique information to be the UID of each node throughout the chapter.

Depth-d Tree. Given any initial graph G_s from a given family, the distributed algorithm must reconfigure the graph into a target graph G_f, such that G_f is a rooted tree of depth d with a unique leader elected at the root.

Apart from studying the running time of our algorithms, measured as their worst-case number of rounds to carry out a given task, we also introduce the following edge complexity measures.

Total Edge Activations. The total number of edge activations of an algorithm is given by $\sum_{i=1}^{T} |E_{ac}(i)|$, where T is the running time of the algorithm.

Maximum Activated Edges. It is defined as $\max_{i\in[T]} |E(i) \setminus E(1)|$, that is, equal to the maximum number of active edges of a round, disregarding the edges of the initial network.

Maximum Activated Degree. The maximum degree of a round, if we again only consider the edges that have been activated by the algorithm. Let $\Delta(G)$ denote the maximum degree of a graph G. Then, formally, the maximum activated degree is equal to $\max_{i\in[T]} deg(D(i) \setminus D(1))$, where the graph difference is defined through the difference of their edge sets.

In this chapter, instead of measuring the maximum activated degree we will focus on preserving the maximum degree of input networks from specific families. For example, one of our algorithms solves the Depth-d Tree problem on any input network and, if the input network has bounded degree, then it guarantees that the degree in any round is also bounded.

2.3.3 Basic Subroutines

We will now provide algorithms that transform initial graphs into graphs with small diameter and which will be used as subroutines in our general algorithms. The first called TreeToStar transforms any initial rooted tree graph into a spanning star in $O(\log n)$ time with $O(n \log n)$ total edge activations and $O(n)$ active edges per round, provided that the nodes have a sense of orientation on the tree (i.e., can distinguish which of their neighbors is "closer" to the root of the tree). In every round, each node activates an edge with the potential neighbor that is its grandparent and deactivates the edge with its parent. This process keeps being repeated by each node until they activate an edge with the root of the tree.

Proposition 2.1. *Let T be any tree rooted at u_0 of depth d. If the nodes have a sense of orientation on the tree, then algorithm TreeToStar transforms T into a spanning star centered at u_0 in $\lceil \log d \rceil \leq \log n$ rounds. TreeToStar has at most $2n - 3$ active edges per round.*

Our next algorithm called LineToCompleteBinaryTree transforms any line into a binary tree in $O(\log n)$ time, with $O(n \log n)$ total edge activations, $O(n)$ active edges per round and the degree of each node is at most 4, provided that the nodes have a common sense of orientation. In each round, each node activates an edge with its grandparent and afterwards it deactivates its edge with its parent. This process keeps being repeated by each node until they activate an edge with the root of the tree or if their grandparent has 2 children.

Proposition 2.2. *Let T be any line rooted at u_0 of diameter d. If the nodes have a sense of orientation on the line, then algorithm LineToCompleteBinaryTree transforms*

T into a binary tree centered at u_0 in $\lceil \log d \rceil \leq \log n$ time. LineToCompleteBinaryTree has at most $2n-3$ active edges per round, $n \log n$ total edge activations and bounded degree equal to 3.

2.3.4 General Strategy for Depth-d Tree

All algorithms developed in this chapter solve the Depth-d Tree problem starting from any connected initial network G_s from a given family. Our aim is to always achieve this in (poly)logarithmic time while minimizing some of the edge-complexity parameters. There is a natural trade-off between time and edge complexity and each of our algorithms makes a different contribution to this trade-off. In particular, by paying for linear degree, our first algorithm manages to be optimal in all other parameters. If we instead insist on bounded degree, then our second algorithm shows that we can still solve Depth-d Tree within an additional $O(\log n)$ factor both in time and total edge activations. Finally, if the bound on the degree is slightly relaxed to (poly)$\log(n)$, our third algorithm achieves $o(\log^2 n)$ time.

All three algorithms are built upon the same general strategy that we now describe. For each of them we choose an appropriate gadget network, which has the properties of being "close" to the target network G_f to be constructed and of facilitating efficient growth. For example, the G_f of our first algorithm is a spanning star and the chosen gadget is a star graph, while the G_f of our second algorithm is a complete binary tree and the chosen gadget is the union of a ring and a complete binary tree spanning that ring (called a *wreath*).

Our algorithms satisfy the following properties. The nodes are always partitioned into committees, where each committee is internally organized according to the corresponding gadget network of the algorithm and has a unique leader, which is the node with maximum UID in that committee. Initially, every node forms its own trivial committee and committees increase their size by competing with nearby committees. In particular, committees select and, if possible, merge with the maximum-UID committee in their neighborhood. Prior to merging, such selections may give rise to pairs of committees, in which case merging is immediate, but also to rooted trees of committees where all selections are oriented towards the root and merging has to be deferred. In the latter case, the winning committee will eventually be the root of the tree, at which point all other committees of the tree will have merged to it. In all cases, merging must be done in such a way that the gadget-like internal structure of the winning committee is preserved. This growth guarantees that eventually there will be a single committee spanning the network. At that point, the leader of that committee (which is always the node with maximum UID in the network) is an elected unique leader. Moreover, the gadget-like internal structure of that committee can be quickly transformed into

the desired target network, due to the by-design close distance between them. For example, in the algorithm forming a star no further modification is required, while in the algorithm forming a complete binary tree, a ring is eliminated from a wreath so that only the tree remains.

Our algorithms are designed to operate in asynchronous phases, with the guarantee that in every phase pairs of committees merge and trees of committees halve their depth. This can be used to show that in all our algorithms a single committee will remain within $O(\log n)$ phases. Each phase lasts a number of rounds which is within a constant factor of the maximum diameter of a committee involved in it, which is in turn upper bounded by the diameter of the final spanning committee. The latter is always equal to the diameter of the chosen gadget as a function of its size. The total time is then given by the product of the number of phases and the diameter of the chosen gadget. For example, in our first algorithm the gadget is a star and the running time (in rounds) is $O(1) \cdot O(\log n)$, in our second algorithm the gadget is a wreath of diameter $O(\log n)$ and the running time is $O(\log n) \cdot O(\log n) = O(\log^2 n)$, while in our third algorithm the gadget is a modified wreath, called ThinWreath, of diameter $o(\log n)$ and the running time is $o(\log n) \cdot O(\log n) = o(\log^2 n)$. Given that every node activates at most one edge per round, the total number of edge activations of our algorithms is within a linear factor of their running time.

2.4 An Edge Optimal Algorithm for General Graphs

Our first algorithm, called `GraphToStar`, solves the Depth-d Tree problem, for $d = 1$. In particular, by using a star gadget it transforms any initial graph G_s into a target spanning star graph G_f. Its running time is $O(\log n)$ and it uses an optimal number of $O(n \log n)$ total edge activations and $O(n)$ active edges per round. Optimality is established by matching lower bounds, presented in Section 2.7.

Algorithm GraphToStar

Each committee $C(u)$ is a star graph where the center node u is the leader of the committee and all other nodes are followers. A committee is anotion we use for a group of nodes that coordinate together and act based on their leader's decision. The leader node of each committee is the node with the greatest UID in that committee. The UID of each committee is defined by the UID of that committee's leader. The winning committee in the final graph, denoted $C(u_{max})$, is the one with the greatest UID in the initial graph. Every node starts as a leader and forms its own committee as a single node. The original edges of G_s are assumed to be maintained until the last round of the algorithm and the nodes can always distinguish them. The algorithm proceeds in phases, where in every phase each committee $C(u)$ executes in one of the following modes, always executing in selection mode in phase 1.

- **Selection:** If $C(u)$ has a neighboring committee $C(z)$ such that $UID_z > UID_u$ and $C(z)$ is not in *pulling mode*, then, from its neighboring committees not in pulling mode, $C(u)$ *selects* the one with the greatest UID; call the latter $C(v)$. It does this, by u first activating an edge e_1 with a potential neighbor in $C(v)$. Then u activates an edge with v, deactivates the previous edge e_1, and $C(u)$ enters either the merging or pulling mode. In particular, if $C(v)$ did not select, then $C(u)$ and $C(v)$ form a pair and $C(u)$ enters the merging mode. If on the other hand $C(v)$ selected some $C(w)$, then $C(u)$ enters the pulling mode. Otherwise, $C(u)$ did not select. If $C(u)$ was selected then it enters the waiting mode, else it remains in the selection mode. If $C(u)$ has no neighboring committees, then it enters the termination mode.

- **Merging:** Given that in the previous phase the leader of $C(u)$ activated an edge with the leader of $C(v)$, each follower x in $C(u)$ activates the edge xv and deactivates the edge xu. The result is that $C(u)$ and $C(v)$ have merged into committee $C(v)$, which remains a star rooted at v now spanning all nodes in $V(C(u)) \cup V(C(v))$. Therefore, $C(u)$ does not exist any more.

- **Pulling:** Given that in the previous phase the leader of $C(u)$ activated an edge with the leader of $C(v)$ and the leader of $C(v)$ activated an edge with the leader of $C(w)$, u activates uw, deactivates uv, and $C(u)$ remains in pulling mode. If, instead, the leader of $C(v)$ did not activate in the previous phase, then $C(u)$ enters the merging mode. On the other hand, given that in the previous phase the leader of $C(u)$ activated an edge with the leader of $C(v)$ and in the current phase, committee $C(v)$ does not exist anymore, this means that v is currently in some committee $C(w)$, and u activates uw and $C(u)$ enters the merging mode.

- **Waiting:** If $C(u)$ has no neighboring committees, $C(u)$ enters the termination mode. If in the previous phase no committee $C(v)$ activated an edge with u, then $C(u)$ enters the selection mode. Otherwise $C(u)$ remains in the waiting mode.

- **Termination:** $C(u)$ deactivates every edge in $E(G_s) \setminus E(C(u))$. In particular, each follower x in $C(u)$ deactivates all active edges incident to it but xu.

Correctness

We are now going to prove the correctness of the algorithm. We will do so by showing that only a single committee is left at the end of the execution of the algorithm. We can guarantee this by showing that only a single committee will enter the termination phase while the rest of them will "die" out by entering the merging phase. Since every committee is a star subgraph and only a single committee will be alive, then the algorithm solves the problem.

Algorithm 1 High level phase transition of each committee in the GraphToStar Algorithm

```
▷state : phase
▷initial state of committee phase C(u) : phase = selection
if phase = selection then
  if there no neighboring committee exists then
    phase = termination
  if there is a neighboring committee C(v) with higher UID not in pulling phase
  then
    if C(v) selected no committee then
      phase = merging
    if C(v) selected another committee then
      phase = pulling
  if there is no neighboring committee with higher UID then
    if no committee C(v) selected C(u) then
      phase = selection
    if another committee C(v) selected C(u) then
      phase = waiting
if phase = merging then
  Terminate
if phase = pulling then
  if the committee I selected is in pulling mode then
    phase = pulling
  if the committee I selected is in merging mode then
    phase = merging
if phase = waiting then
  if no committee activated an edge with me then
    phase = selection
if phase = termination then
  Terminate
```

Lemma 2.3. *Algorithm GraphToStar solves Depth-1 Tree.*

Proof. It suffices to prove that in any execution of the algorithm, one committee eventually enters the termination mode and that this committee can only be $C(u_{max})$. If this holds, then by the end of the termination phase $C(u_{max})$ forms a spanning star rooted at u_{max} and u_{max} is the unique leader of the network. This satisfies all requirements of Depth-1 Tree.

A committee *dies* (stops existing) only when it merges with another committee by entering the merging mode. First observe that there is always at least one *alive* committee. This is $C(u_{max})$, because entering the merging mode would contradict maximality of u_{max}. We will prove that any other committee eventually dies or grows, which due to the finiteness of n will imply that eventually $C(u_{max})$ will be the only alive committee.

In any phase, but the last one which is a termination phase, it holds that every alive committee $C(u)$ is in one of the selection, merging, pulling, and waiting modes. If $C(u)$ is in the merging mode, then by the end of the current phase it will have died by merging with another committee $C(v)$. It, thus, remains to argue about committees in the selection, pulling, and waiting modes.

We first argue about committees in the pulling mode. Denote their set by $\mathcal{C}_{pull}$. Observe that, in any given phase, the committees in pulling mode form a forest F, where each $C(u) \in \mathcal{C}_{pull}$ belongs to a pulling tree T of F. Any such pulling tree mimicks the execution of the TreeToStar algorithm (from Proposition 2.1) on the leaders of committees $C(u)$ and satisfies the invariant that its root committee C_r is always in the waiting mode and C_r's children are in the merging mode. In every phase, C_r's children merge with C_r and their children become the new children of C_r and enter the merging mode. It follows that all non-root committees in T will eventually merge with C_r. Thus, all committees in pulling mode eventually die.

It remains to argue about committees in the selection and waiting modes. We start from the waiting mode. Any committee $C(u)$ in waiting mode is a root of either a pulling tree in the forest F or of a star of committees in which all leaf-committees are merging with $C(u)$. In both cases, $C(u)$ eventually exits the waiting mode and enters the selection mode. This happens as soon as all other committees in its pulling tree or star have merged to it, thus $C(u)$ has grown upon its exit.

Now, a committee $C(u)$ in the selection mode can enter any other mode. As argued above, if it enters the merging or pulling modes it will eventually die and if it enters the waiting mode it will eventually grow. Thus, it suffices to consider the case in which it remains in the selection mode indefinitely. This can only happen if all current and future neighboring committees of $C(u)$, including the ones to eventually replace neighbors in pulling mode, have an UID smaller than UID_u. But each of these must have selected a neighboring $C(w)$, such that $UID_w > UID_u$, otherwise it would have selected $C(u)$. Any such selection results in $C(w)$ (or a z, such that $UID_z > UID_w$ in case w belongs to a tree) becoming a neighbor of $C(u)$, thus contradicting the indefinite local maximality of UID_u.

□

Time Complexity

Let us move on to proving the time complexity of our algorithm. At the beginning, we are going to ignore the number of rounds within a phase, and we are just going to study the maximum number of phases before a single committee is left. We define $|C(u)_s|$ to be the *size* of committee $C(u)$ in phase s, which is the number of nodes in committee $C(u)$ in phase s.

Lemma 2.4. *Consider committee $C(u)$ that is in waiting mode between phases s and $s+j$. If the size of every committee in phase s is at least 2^k, then the size of committee $C(u)$ once it enters the selection mode in phase $s+j+1$ is at least 2^{k+j-2}.*

Proof. Any committee $C(u)$ in waiting mode is a root of either (i) a pulling tree in the forest F or (ii) a star of committees in which all leaf-committees are merging with $C(u)$.

For case (i): root committee $C(u)$ is always in waiting mode and every other committee $C(v)$ of T is either in pulling or merging mode. It follows that all non-root committees $C(v)$ in the pulling tree will eventually merge with $C(u)$ in some phase $s+j$. W.l.o.g. assume that every committee $C(v)$ that belongs to the pulling tree T entered pulling or merging mode in phase s and every committee $C(v)$ will have merged with committee $C(v)$ by phase $s+j$. Every committee $C(v)$ will stay in pulling mode for $i < j$ phases and in merging mode for 1 phase. Consider the leaders v of every committee $C(v)$ and note that while in pulling mode, the leaders are mimicking the execution of the TreeToStar algorithm, where the leader of $C(u)$ is the root of the tree, and the leaders of $C(v)$ are the non-root nodes of the tree. We know by Proposition 2.1, that the running time of the algorithm is $\log d$, where d is the depth of the tree. Thus, if every committee $C(v)$ enters the pulling mode in phase s and the last committee $C(v)$ to exit the pulling mode is in phase $s+i$, $s+i-s = \log d \implies i = \log d$. This means that the depth of tree T is 2^i. Since the depth of the pulling tree T is 2^i, the tree T must contain at least 2^i committees. Additionally note that after the last committee $C(v)$ exits the pulling mode is in phase $s+k$, in phase $s+k+1$ it enters the merging mode and in phase $s+k+2$ every committee $C(v)$ has merged with committee $C(u)$. Thus, $s+i+2 = s+j \implies i = j-2$ and the size of $C(u)$ in phase $s+j+1$ is $|C(u)_{s+j+1}| \geq 2^k * 2^i = 2^{k+j-2}$.

For case (ii): root committee $C(u)$ is in waiting mode and has at least one leaf committee in phase s. After the leaf committee merges in 1 phase, committee $C(u)$ has size $|C(u)_{s+1}| \geq |C(u)_s| + |C(u)_s| =\geq 2^k + 2^k = 2^{k+1}$. □

Lemma 2.5. *If committee $C(u)$ stays in the selection mode for $p \geq 4$ consecutive phases, then $C(u)$ has a neighboring committee $C(v) \in \mathcal{C}_{pull}$ that belongs to a pulling tree T for at least p phases.*

Proof. Let us assume that committee $C(u)$ stays in the selection mode for $p \geq 4$ consecutive phases while having a neighbor $C(v)$ that does not belong to pulling tree T.

- If $C(v)$ does not belong to a pulling tree in phase k, then it cannot be in pulling mode.

- If $C(v)$ is in selection mode in phase k and $C(v)$ does not select $C(u)$ and $C(u)$ does not select $C(v)$, then $C(v)$ has a neighbor $C(w)$ where $UID_w > UID_v > UID_u$ and $C(v)$ selected $C(w)$. Then $C(v)$ enters the merging mode in phase $k+1$ and gets merged with $C(w)$. In phase $k+2$ committee $C(w)$ becomes a neighbor of $C(u)$ and $C(w)$ enters the selection mode. Therefore, since $UID_w > UID_u$, $C(u)$ would select $C(w)$ in phase $k+2$, and enter either the pulling or merging mode. Thus, a contradiction.

- If $C(v)$ is in waiting mode in phase k, it cannot be the root of a pulling tree, and is the root of a star. Therefore in phase $k+1$ it will enter the selection mode and based on the analysis of the previous paragraph, in phase $k+3$ $C(u)$ will exit the selection mode. Thus, a contradiction.

□

Lemma 2.6. *Let us assume that the minimum size of a committee in phase s is 2^k. If committee $C(u)$ stays in the selection mode from phase s to phase $s+p$, where $p \geq 4$ and those p phases are consecutive, then in phase $s+p+1$ it will select or get selected by a committee $C(v)$ of size at least 2^{k+p-2}.*

Proof. From Lemma 2.5 it follows that, since $C(u)$ is in the selection mode for at least 4 phases, there exists a neighbor $C(v)$ that belongs to a pulling tree T. W.l.o.g. assume that $C(w)$ is the root of the pulling tree T and $C(w)$ has been in waiting mode between phases s and $s+p$. Note also that in phase $s+p+1$, committees $C(u)$ and $C(w)$ are neighboring committees and both are in selection mode. Thus, $C(u)$ will exit the selection mode in phase $s+p+1$, because either $C(u)$ will select $C(w)$ or $C(w)$ will select $C(u)$. Since $C(w)$ was in waiting mode for p phases, the size of $C(w)$ is at least 2^{k+p-2} (based on Lemma 2.4). □

Lemma 2.7. *Assume that the minimum size of every committee in phase s is 2^k and that every committee will have exited the selection mode in phase $s+p$ at least once. The size of all winning committees (committees that still exist) in phase $s+p+1$ is at least 2^{k+p-2}.*

Proof. Trivially, if $p \leq 4$ the winning committee has size at least 2^{k+1} in phase $p+1$ since it has merged with at least one other committee. From Lemma 2.6 it follows that if $p \geq 4$ the winning committee between $C(w)$ and $C(u)$ will have size at least 2^{k+p-2} in phase $s+p+1$. □

Lemma 2.8. *After $O(\log n)$ phases, there is only a single committee left in the graph.*

Proof. We trivially assume that committee $C(u_{max})$ has size $|C(u_{max})|_1| = 1$ in phase 1. Based on Lemma 2.7, after $O(\log n)$ phases, $C(u_{max})$ has size $|C(u_{max})_{O(\log n)}| \geq$

$2^{1+O(\log n)-c} \geq 2^{\log n} \geq n$. Therefore, committee $C(u_{max})$ must contain every single node of G. □

Lemma 2.9. *Each phase consists of at most* 2 *rounds.*

Proof. Based on the description of the algorithm, the selection phase lasts 2 rounds and the rest of the phases lasts 1 round. □

Edge Complexity

It is very simple to prove the edge complexity for the algorithm. Note that in each round i each node activates at most 1 edge and based on Lemma 2.8 the algorithm runs for $O(\log n)$ phases which means that there are $O(n \log n)$ total edge activations. Furthermore, if a node had activated an edge u in round i, and it activates another edge v in round $i+1$, then it deactivates edge u. Therefore, each node cannot have more than 2 active edges that it has activated itself at any time and since we have n nodes in the network, there can ever be at most $2n$ active edges per round. Since the structure of every committee is a star, the maximum activated degree is $O(n)$.

Theorem 2.10. *For any initial connected graph G_s, the* GraphToStar *algorithm solves the Depth-1 Tree problem in $O(\log n)$ time with at most $O(n \log n)$ total edge activations, $O(n)$ active edges per round and $O(n)$ maximum activated degree.*

2.5 Minimizing the Maximum Degree on General Graphs

In this section we will create an algorithm that minimizes the maximum activated degree to a constant but has $O(\log^2 n)$ running time and $O(n \log^2 n)$ total edge activations.

For this algorithm, our committees must have at least $\Omega(\log n)$ diameter in order to have a constant degree and therefore merging two different committees in constant time while keeping a specific structure proves to be complicated. The new gadget of our committees is going to be a graph we call *wreath*. A wreath graph is a graph that has both a ring subgraph and a complete binary tree subgraph. We are going to use the edges of the ring subgraph to merge committees and the binary tree subgraph to exchange information between the nodes of the graph. First, let us define the structure of the wreath graph.

Definition 2.11 (Wreath graphs)**.** A graph $D = (V, E)$ belongs to the class of *wreath* graphs if it has two subgraphs $D_r = (V, E_r)$ and $D_b = (V, E_b)$, where D_r belongs to the class of ring graphs, D_b belongs to the class of complete binary tree graphs, and $E = E_r \cup E_b$.

The $O(\log n)$ diameter that the wreath graph possesses will allow the leaders of committees $C(u)$ to communicate with neighboring committees $C(v)$ in $O(\log n)$ time.

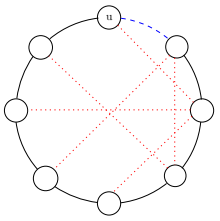

FIGURE 2.1: A wreath graph with 8 nodes. The ring subgraph consists of the normal (black) and dashed (blue) edges. The complete binary tree consists of the dotted (red) and dashed (blue) edges.

Additionally, the merging phase of each pair of committees will require only $O(\log n)$ time. The algorithm is almost identical to the `GraphToStar` as far as the high level strategy is concerned. Committees select neighboring committees and merge with them. The main difference is that when a tree with root v is formed, we cannot use the pulling mode since this would increase the degree significantly. We provide an example of two committees merging in Fig. 2.2. In this example, committee $C(u)$ is merging with committee $C(v)$ and the merging happens through nodes x and y, see Fig. 2.2(a). The committees on each tree merge in a single ring that includes all committees in $O(1)$ time (ring merging mode), see Fig. 2.2(b),2.2(c). After this, v deactivates one of its incident edges in order to create a line subgraph, see figure 2.2(d). Once this happens, each node on the line executes an asynchronous version of the LineToCompleteBinaryTree subroutine in $O(\log n)$ time using the orientation of the new ring, where root v is the root of the line. Once the subroutine is finished, the complete binary tree subgraph of the wreath graph is ready. Therefore we have managed to merge a tree graph of multiple committees into a single committee. Fig. 2.2 does not include the asynchronous version of the LineToCompleteBinaryTree subroutine since it is quite involved to illustrate.

Algorithm GraphToWreath

The structure of each committee/node is the same as the `GraphToStar` algorithm apart from the fact that each committee $C(u)$ is a wreath graph. Every node is able to distinguish between the edges of the binary tree and the edges of the ring by marking them and it can also distinguish its clockwise neighbor and counterclockwise neighbor on the ring. Our algorithm proceeds in phases, where in every phase each committee $C(u)$ executes in one of the following modes, always executing in selection mode in phase 1.

- **Selection:** If $C(u)$ has a neighboring committee $C(z)$ such that $UID_z > UID_u$ and $C(z)$ is not in *Ring Merging mode* or *Tree Merging mode* then, from its neighboring committees not in ring merging or tree merging mode, $C(u)$ *selects*

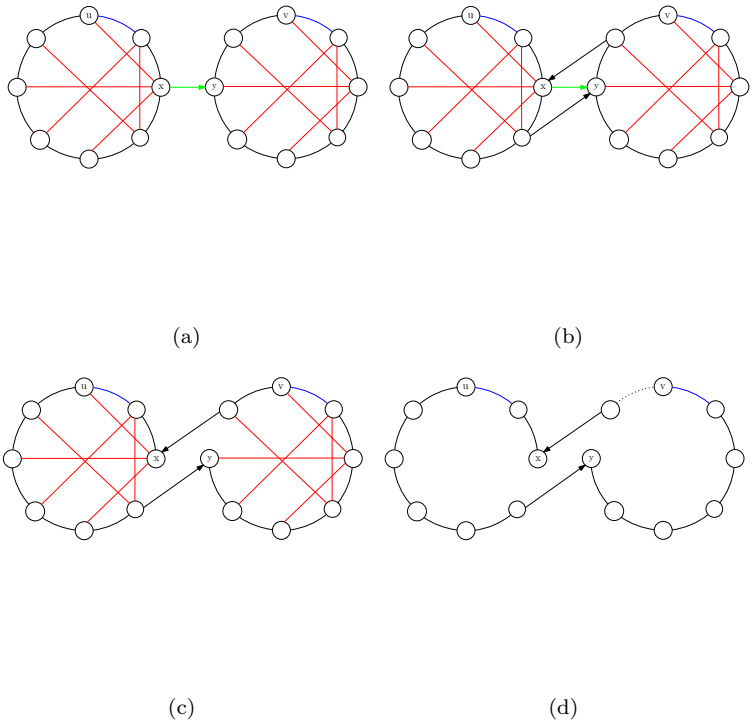

FIGURE 2.2: Example where committee $C(u)$ is merging with committee $C(v)$ and the merging happens through nodes x and y. Figure (a) shows the initial connections. Figure (b) shows that ring merging process where x and y activate edges with the counterclockwise neighbors of each other. Figure (c) shows the deactivation of edges from x and y in order to form the cycle which includes the black and blue edges. Figure (d) shows the that node v deactivates its incident edge (dotted line) in order to turn the cycle into a line where the asynchronous version of the LineToCompleteBinaryTree subroutine will be executed.

the one with the greatest UID; call the latter $C(v)$. If $C(u)$ selected $C(v)$ or $C(u)$ was selected, $C(u)$ enters the Ring Merging mode. If $C(u)$ did not select anyone and it was not selected by anyone, it stays in the selection mode. If $C(u)$ has no neighboring committees, $C(u)$ enters the termination mode.

- **Ring Merging:** Given that in the previous phase, $C(u)$ selected $C(v)$, committee $C(u)$ merges its ring component with the ring component of $C(v)$ by the following method: Let $k \in C(u)$ and $l \in C(v)$, such that edge kl is active. k activates an edge with the clockwise neighbor of l, call it l_1, and l activates an edge with the clockwise neighbor of k, call it k_1. Then they deactivate edges kk_1, ll_1, and kl. The two rings have now merged into a single ring.

  Given that in the previous phase, $C(u)$ was selected by $C(k)$, committee $C(k)$ merges its ring component with the ring component of $C(u)$. $C(u)$ enters the tree merging mode.

- **Tree Merging:** Every node x in $C(u)$ executes one round of an asynchronous version of the LineToCompleteBinaryTree algorithm, which extends the LineToCompleteBinaryTree algorithm with extra wait states. If there exists node x that has not terminated the asynchronous LineToCompleteBinaryTree algorithm, $C(u)$ stays in the Tree Merging mode. If all nodes x have terminated the asynchronous LineToCompleteBinaryTree algorithm, all nodes x have now merged with committee $C'(u)$ whose leader is the root of the complete binary tree and $C'(u)$ enters the selection mode. $C(u)$ does not exist anymore.
- **Termination:** Each follower x in $C(u)$ deactivates every edge apart from the edges that define the spanning complete binary tree subgraph.

Algorithm 2 High level phase transition of each committee in the GraphToWreath Algorithm

```
▷state : phase
▷initial state of committee phase C(u) : phase = selection
if phase = selection then
  if there no neighboring committee exists then
    phase = termination
  if there is a neighboring committee C(v) with higher UID not in tree merging or
  ring merging phase then
    phase = ringmerging
  if there is no neighboring committee with higher UID then
    if no committee C(v) selected C(u) then
      phase = selection
    if another committee C(v) selected C(u) then
      phase = ringmerging
if phase = ringmerging then
  phase = treemerging
if phase = treemerging then
  if tree merging subroutine has not finished then
    phase = treemerging
  if tree merging subroutine has finished then
    if C(u) is leader then
      phase = selection
    if C(u) is not leader then
      Terminate
if phase = termination then
  Terminate
```

2.5.1 Low Level Description of Modes

In this subsection, we are going to describe the low level details of each mode since the communication process is much more complicated than the `GraphToStar` algorithm.

Selection. Consider committee $C(u)$. Each follower x in committee $C(u)$ sends a message $\{myUID_x, maxNeighborUID, maxNeighborDiameter\}$ to its leader u via the binary tree subgraph, see Fig. 2.3. Variable $myUID_x$ contains the UID of node x, $maxNeighborUID$ contains the UID of the neighboring committee with the greatest UID among all neighboring committees that x has an edge with, and $maxNeighbor-Diameter$ contains the diameter of that committee.

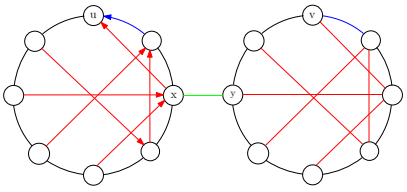

FIGURE 2.3: Every follower in $C(u)$ sends a message with the information of its neighboring committees to leader u via the complete binary tree. For example, follower x sends the information for committee $C(v)$

.

After committee leader u receives all triplets, u knows the UID of all neighboring committees. If $\exists\ maxNeighborUID > UID_u$, $C(u)$ selects the neighboring committee $C(v)$ with the greatest $maxNeighborUID$ and broadcasts a message to x to initiate the connection with that committee. Since, it is possible that multiple followers x sent the same $maxNeighborUID$, u picks the one with the greatest UID_x. If $\nexists$ $maxNeighborUID > UID_u$, committee $C(u)$ does not select another committee. Either way, after the selection, u waits to see whether another committee has selected $C(u)$. Committee leader u knows the maximum waiting time since it just received the maximum diameter of all neighboring committees.

After follower x receives the initiation message, it sends a connection message to the leader v of the neighbouring committee $C(v)$ via followers x and y though the binary tree subgraphs. See Fig. 2.4. After leader v receives all possible requests, it sends back an approval message to all nodes y with a timestamp that defines in which round the merging should happen. See Fig. 2.5.

Therefore every committee $C(u)$ can understand which committee $C(v)$ it has selected and whether any committees $C'(v)$ have selected $C(u)$. This means that $C(u)$ knows which mode it should enter after the selection phase.

Ring Merging. Assume that multiple committees $C(v_1), C(v_2), ..., C(v_i)$ for $i = 1, ..., n-1$ have selected committee $C(u)$ in the selection phase, via followers $y_1, y_2, ..., y_i$ respectively, whose neighbour $x \in C(u)$ will initiate the connection. See Fig. 2.6(a) for

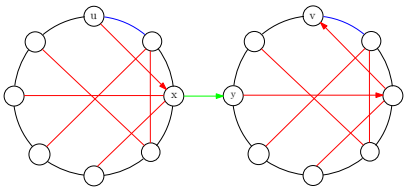

FIGURE 2.4: Leader u sends a message to follower x to initiate a connection with committee v. Follower x sends the request to follower y who propagates it to leader v

.

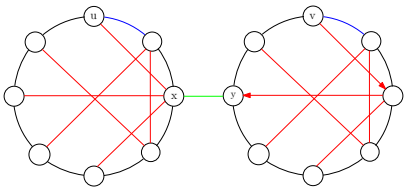

FIGURE 2.5: Leader v sends the approval message to follower y to initiate the merging with committee u

.

an example. Followers $y_1, y_2, ..., y_i, x$ execute the following steps in order to complete the ring merging mode.

- Follower x sends a message to followers $y_1, y_2, ..., y_i$ to rearrange themselves into an *inner-circle* by activating edges $\{x, y_1\}$, $\{y_1, y_2\}, \{y_2, y_3\}, ..., \{y_{i-1}, y_i\}, \{y_i, x\}$ and deactivating edges $\{y_1, x\}, \{y_2, x\}, ..., \{y_i, x\}$. See Fig. 2.6(b).
- Each follower activates an edge with the clockwise neighbor of its inner-circle outgoing neighbor. See Fig. 2.6(c).
- Each follower deactivates an edge with its clockwise neighbor, as well as the edges of the inner-circle. See Fig. 2.6(d).

Note that the orientation of the new ring is the same as the orientation of committee $C(u)$ and all nodes in the committee have the same orientation.

Tree Merging. Note that we cannot use the LineToCompleteBinary tree algorithm from section 2.3.3 to merge the tree component of the committees since that algorithm assumes that every node starts the execution at the same time. But in our case, we

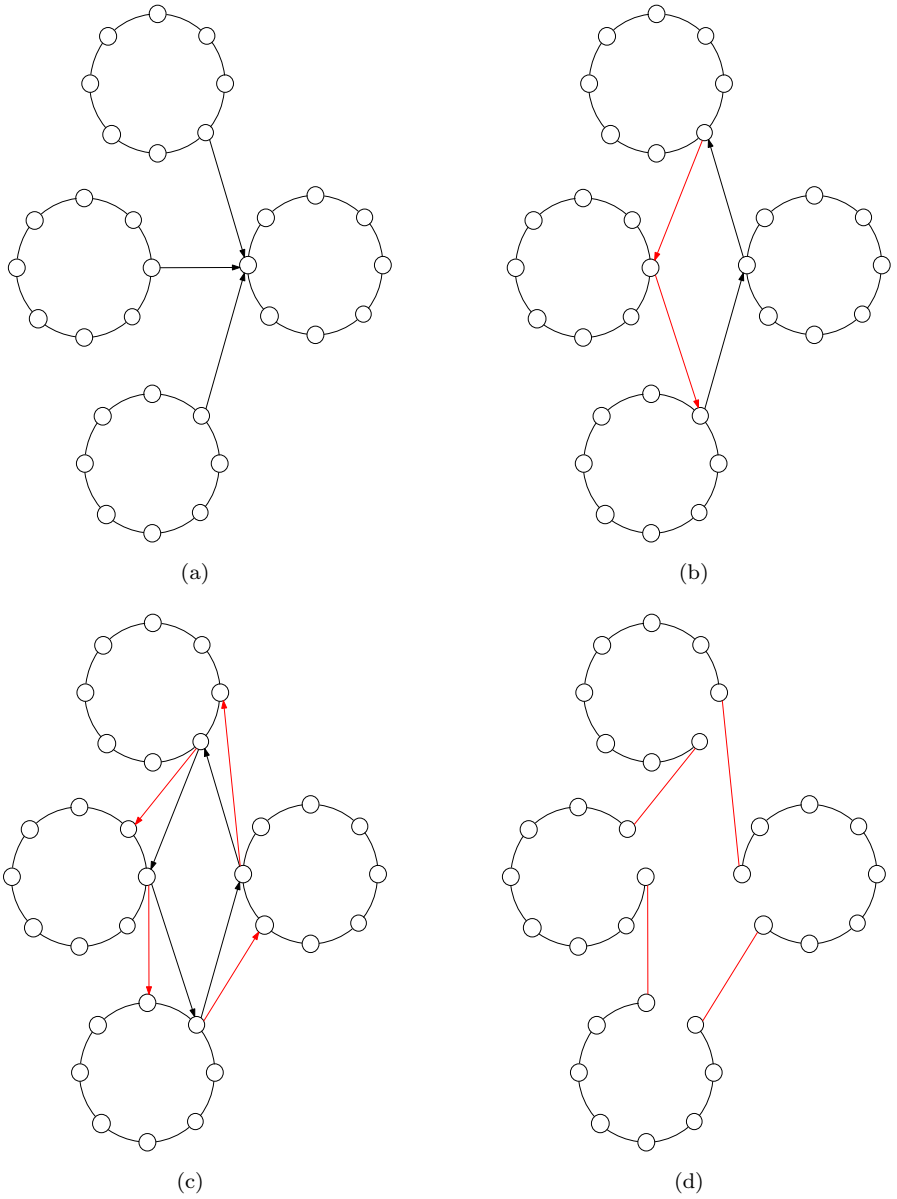

FIGURE 2.6: Example where 3 committees $C(v_1), C(v_2), C(v_3)$ have selected committee $C(u)$. Figure (a) shows the initial connection. In figure (b) committees rearrange themselves into an inner-circle. In figure (c) each committee activates an edge with the clockwise neighbor of its inner-circle outgoing neighbor. In figure (d) each committee deactivates an edge with its clockwise neighbor(based on the committee orientation), as well as the edges of the inner-circle.

have multiple committees that are merging together with different sizes and therefore the nodes are not synchronized. Thus, we introduce an asynchronous version of the algorithm where nodes can start the execution at different rounds.

Every node x executes the asynchronous LineToCompleteBinaryTree algorithm which works as follows. If node x was a committee leader, then $leader_x = true$ else $leader_x = false$. The acronym EA stands for Edge Activations and DEA stands for Edge Deactivations.

Algorithm 3 Asynchronous LineToCompleteBinaryTree

▷*state* : EA, DEA, *awake*, *leader*
▷*initial state of node* : $EA = 0$, $DEA = 0$, $Awake = false$
if *node receives awake signal* OR $leader = true$ **then**
  $Awake = true$
if $awake = true$ **then**
  Broadcast awake
  if *grandparent has only* 1 *child* **then**
    if $EA_{my} = DEA_{my} = EA_{father} = DEA_{father}$ **then**
      Activate edge with grandparent
      $EA_{my}++$
    if $EA_{my} = DEA_{my} + 1 = EA_{child}$ **then**
      Deactivate edge with parent
      $DEA_{my}++$

We are going to give an intuition on how this algorithm works. First, the leader of each committee broadcasts an awake signal to its own committee. Once a node awakes, it starts executing the asynchronous LineToCompleteBinaryTree algorithm. Since nodes have different waking points, we cannot use the synchronous LineToCompleteBinaryTree that requires synchronized clocks from each node. Therefore, we are going to use other properties that are present for every node in the synchronous LineToCompleteBinaryTree which are: (i) Every node x has the same total number of activations as its parent. (ii) Every node has the same number of total activations as total deactivations. The asynchronous version tries to mimic that by having every node activate an edge, only when its parent has the same total number of edge activations as itself. Similarly, for the deactivations, every node checks that its child has the same deactivations as itself before deactivating an edge. This way, the synchronous LineToCompleteBinaryTree is simulated by the asynchronous version.

Correctness

We are now going to prove the correctness of the algorithm. We will do so by showing that only a single committee is left at the end of the execution of the algorithm. We can guarantee this by showing that only a single committee will enter the termination phase while the rest of them will "die" out by entering the merging phase. Since

every committee is a star subgraph and only a single committee will be alive, then the algorithm solves the problem.

Lemma 2.12. *Algorithm GraphToWreath solves Depth-*$\log n$ *Tree.*

Proof. It suffices to prove that in any execution of the algorithm, one committee eventually enters the termination mode and that this committee can only be $C(u_{max})$ where u_{max} is the highest UID in the network. If this holds, then by the end of the termination phase $C(u_{max})$ forms a complete binary spanning tree rooted at u_{max} and u_{max} is the unique leader of the network. This satisfies all requirements of Depth-$\log n$ Tree.

A committee *dies* only when it merges with another committee by entering the tree merging mode. First observe that there is always at least one *alive* committee. This is $C(u_{max})$, because when it enters the tree merging mode, it is always the root of the complete binary tree. We will prove that any other committee eventually dies or grows, which due to the finiteness of n will imply that eventually $C(u_{max})$ will be the only alive committee.

In any phase, but the last one which is a termination phase, it holds that every alive committee $C(u)$ is in one of the selection, ring merging, and tree merging modes. If $C(u)$ is in the ring merging mode then it will enter the tree merging mode and if its leader is not the root of the complete binary tree, then by the end of the current phase it will have died by merging with another committee $C'(u)$. It, thus, remains to argue about committees in the selection mode.

Now, a committee $C(u)$ in the selection mode can enter the tree merging mode. As argued above, if it enters the ring merging and tree merging modes in sequence it will either die or it will eventually grow. Thus, it suffices to consider the case in which it remains in the selection mode indefinitely. This can only happen if all current and future neighboring committees of $C(u)$ have an UID smaller than UID_u. But each of these must have selected a neighboring $C(w)$, such that $UID_w > UID_u$, otherwise it would have selected $C(u)$. Any such selection, results in $C(w)$ becoming a neighbor of $C(u)$, thus contradicting the indefinite local maximality of UID_u. □

Time Complexity

Let us move on to proving the time complexity of our algorithm. At the beginning, we are going to ignore the number of rounds within a phase, and we are just going to study the maximum number of phases before a single committee is left.

Lemma 2.13. *After* $O(\log n)$ *phases, there is only a single committee left in the graph.*

Proof. Note that there is a direct correspondence between the modes in the `GraphToWreath` algorithm and the `GraphToStar` algorithm.

Both selection modes are used to decide the selections between the neighboring committees. The difference between the two algorithms is that each selection phase

has a different running time. In particular, The `GraphToStar` selection phase required 2 rounds while the selection phase of the `GraphToWreath` requires $O(\log n)$ rounds due to the diameter of the Wreath graph that each committee has. Therefore Lemma 2.5 that talks about the selection waiting time still holds.

The ring mode is always an intermediate phase between the selection phase and the tree merging phase that lasts for $O(1)$ rounds. The purpose of this mode is to turn the tree T created by the committees in the selection phase into a cycle so that the LineToCompleteBinaryTree subroutine can work. The pulling mode in the `GraphToStar` implements the TreeToStar subroutine, while the tree merging mode in the `GraphToWreath` implements the asynchronous version of the LinetoCompleteBinaryTree. Both subroutines are used to merge the Trees T of depth t created by the committees in $O(\log t)$ time and recall from the basic subroutines subsection that the TreeToStar and the LineToCompleteBinaryTree have the same running time. Therefore both algorithms require the same amount of phases. Therefore Lemmas 2.4, 2.6 and 2.7 that show the growth of each committee still hold.

Note that there is no merging or waiting mode in the `GraphToWreath` since those modes have also been implemented by the merging tree mode.

Since all modes that have been implemented in the `GraphToWreath` have equivalent modes in the `GraphToStar` with similar running times and growths for the committes, the `GraphToWreath` algorithm requires at most $O(\log n)$ phases. □

Lemma 2.14. *Each phase in the GraphToWreath algorithm, requires at most* $O(\log n)$ *rounds.*

Proof. First, we argue that the selection phase requires $O(\log n)$ rounds since each committee $C(u)$ has to exchange information with its neighboring committees in order to decide which committee $C(w)$ it is going to merge with and whether any other committee $C(v)$ will decide to merge with $C(u)$. This requires time that is upper bounded by the diameter of each committee. Based on the low level description of the selection mode, the leader of committee $C(u)$ learns the UID of every neighboring committee in $\log d$ rounds and initiates the connection with the chosen neighboring committee $C(w)$ in another $\log d$ rounds, where d is the diameter of committee $C(u)$. Another $\log d_w$ rounds are required in order for committee $C(w)$ to accept and initiate the connection with committee $C(u)$, where d_w is the diameter of $C(w)$. Since $\log d \leq \log n$ and $\log d_w \leq \log n$, the selection mode requires $O(\log n)$ rounds.

The ring merging phase requires $O(1)$ rounds since every committee has to merge its ring component with committees $C(v)$ and the running time does not depend on the size of each committee participating.

Each tree merging mode implements one round of the asynchronous LineToCompleteBinaryTree. Note here that the asynchronous version of this algorithm has the

same running time as the synchronous version if we consider round 0 to be the first round in which all nodes are awake. Additionally, since every committee leader broadcasts the awake message to its own committee, the time needed for all nodes to be awake is $\log(\max d) < \log n$. Thus, the running time of the asynchronous LineToCompleteBinaryTree is $O(\log n)$.

□

Edge Complexity

The analysis for the total edge activations is simple. The algorithm runs for $O(\log^2 n)$ rounds and each node activates at most 1 edge per round. Therefore the total edge activations are $O(n \log^2 n)$.

Let us consider the maximum incident edges that a node can have, excluding the edges of the initial graph. Each node has up to 2 edges for the ring component of the wreath and 2 for the binary tree component of the wreath graph. Based on the low level description of the `GraphToWreath` algorithm, a node can have 1 active edge used for the ring merging phase. Additionally, it can have 2 active edges for the execution of the LineToCompleteBinaryTree. Therefore the number of active edges per round are $O(n)$ and the maximum degree of each node is $7 + c$, where c is the degree of each node in the original graph.

Theorem 2.15. *For any initial connected graph with constant degree, the GraphToWreath algorithm solves Depth-*$\log n$ *Tree problem in* $O(\log^2 n)$ *time with* $O(n \log^2 n)$ *total edge activations,* $O(n)$ *active edges per round and* $O(1)$ *maximum activated degree.*

2.6 Trading the Degree for Time

In this section, we provide another algorithm aiming at $O(\frac{\log n}{\log \log n})$ time for the merging but we are going to allow the maximum degree to reach $O(\log^2 n)$. This requires a new graph for the committees where the diameter of the shape is $O(\frac{\log n}{\log \log n})$, so that the communication within the committees is $O(\frac{\log n}{\log \log n})$ and a new way to merge the committees in $O(\frac{\log n}{\log \log n})$ time. For this algorithm only, we also make the assumption that all nodes know the size of the initial graph. This yields an interesting open problem on whether we can modify the algorithm so that it will not require knowledge of the initial network.

The new graph is very similar to the Wreath graph and we call it *ThinWreath*. The main difference is that instead of having a complete binary tree component, it has a complete polylogarithmic degree tree component with diameter $O(\frac{\log n}{\log \log n})$. The $O(\frac{\log n}{\log \log n})$ diameter that the ThinWreath graph possesses will allow the leaders of neighboring committees to communicate in $O(\frac{\log n}{\log \log n})$ time.

Algorithm GraphToThinWreath

The structure of each committee is the same as in `GraphToStar` algorithm, apart from the fact that each committee $C(u)$ is a ThinWreath graph. We also assume that the nodes know the size of the initial graph. Our algorithm proceeds in phases, where in every phase each committee $C(u)$ executes in one of the following modes, always executing in selection mode in phase 1.

- **Selection:** If $C(u)$ has a neighboring committee $C(z)$ such that $UID_z > UID_u$ and $C(z)$ is in selection mode, then, $C(u)$ *selects* its neighboring committee with the greatest UID; call the latter $C(v)$. If $C(u)$ was selected by another committee, $C(u)$ enters the Matchmaker mode. If $C(u)$ was not selected and $C(u)$ selected $C(v)$, $C(u)$ enters the Matched mode. If $C(u)$ did not select anyone and it was not selected by anyone, it stays in the selection mode. If $C(u)$ has no neighboring committees, it enters the termination mode.

- **Matchmaker:** If multiple committees had selected $C(u)$ in the previous phase, committee $C(u)$ matches those committees in pairs. If the number of committees that selected $C(u)$ is odd, one committee is matched with $C(u)$. $C(u)$ enters the Matched mode.

- **Matched:** If committee $C(u)$ selected committee $C(v)$ in the last selection phase, committee $C(u)$ is matched with another committee. Committee $C(u)$ enters the Ring Merging mode.

- **Ring Merging:** Given that in the previous phase, $C(u)$ was matched with $C(v)$, committee $C(u)$ merges its ring component with the ring component of $C(v)$ where the winning committee is $C(u)$ if $UID_u > UID_v$, otherwise $C(v)$ is the winning committee. Either way, committee $C(u)$ enters the Leader Merging mode.

- **Leader Merging:** Given that in the previous mode, committee $C(u)$ lost to committee $C(w)$, the leader of $C(u)$ activates an edge with the leader of $C(w)$. If committee $C(w)$ has lost to some other committee $C(z)$ in the previous phase, $C(u)$ enters the Tree Merging mode. If $C(u)$ did not lose to any other committee, $C(u)$ enters the Tree Merging mode where u is the root.

- **Tree Merging:** The leader of $C(u)$ executes one round of the asynchronous LineToCompletePolylogarithmicTree algorithm, which is similar to the asynchronous LineToCompleteBinaryTree algorithm with a termination criterion of $\log n$ children instead of 2. If there exists node x that has not terminated the asynchronous LineToCompletePolylogarithmicTree algorithm, $C(u)$ stays in the Tree Merging mode. If all nodes x have terminated the asynchronous LineToCompletePolylogarithmicTree algorithm, all nodes $x \in C(u)$ have now merged with committee

$C'(u)$ whose leader of $C(u)$ is the root of the complete polylogarithmic tree and $C'(u)$ enters the selection mode. Committee $C(u)$ does not exist anymore.

- **Termination:** Each follower $x \in C(u)$ deactivates every edge apart from the edges that define the complete polylogarithmic spanning tree subgraph.

Algorithm 4 High level phase transition of each committee in the GraphToThinWreath Algorithm

```
▷state : phase
▷initial state of committee phase C(u) : phase = selection
if phase = selection then
  if there no neighboring committee exists then
    phase = termination
  if there is a neighboring committee C(v) with higher UID not in tree merging or
  ring merging phase then
    phase = matchmaker
  if there is no neighboring committee with higher UID then
    if no committee C(v) selected C(u) then
      phase = selection
    if another committee C(v) selected C(u) then
      phase = Matched
if phase = matchmaker then
  phase = matched
if phase = matched then
  phase = ringmerging
if phase = ringmerging then
  phase = leadermerging
if phase = leadermerging then
  phase = treemerging
if phase = treemerging then
  if tree merging subroutine has not finished then
    phase = treemerging
  if tree merging subroutine has finished then
    if C(u) is root then
      phase = selection
    if C(u) is not root then
      Terminate
if phase = termination then
  Terminate
```

2.6.1 Low Level Description of the Modes

We will now describe the low level operation of the modes. The selection and ring merging modes are identical to the equivalent modes of the `GraphToWreath` algorithm and therefore will not be described here.

Matchmaker. Before we begin the description of this mode, we would like to give some insight on the Matchmaker/Matched modes and what they are trying to achieve. After the selection mode, the graph of committees consists of directed trees, directed lines and pairs. We want to break up the directed trees into lines and pairs since directly merging the committees using the directed trees might result in having a final committee with linear degree due to the structure of the directed tree. We break up the committees by pairing up the multiple committees $C(v)$ that have selected committee $C(u)$. The difficulty here arises from the fact that committees $C(v)$ are not neighbours and they have to use committee $C(u)$ in order to become neighbours by activating edges on $C(u)$. While doing this, we have to make sure that these edge activations don't violate the maximum degree of $O(\log n)$.

In this mode, we know that at least one committee $C(v)$ has selected committee $C(u)$. Leader u sends a synchronisation message with a timestamp to all leaders v which dictates when the Matched mode algorithm should begin. This timestamp is equal to $3 \cdot d$ where d is the diameter of the neighbouring committee with the highest UID among all neighbouring committees of $C(u)$. This guarantees that the message can reach every leader v in $2 \cdot d$ time and d time for the v leaders to send the message back to their followers. After this, committee $C(u)$ enters the Matched mode.

Matched. In this mode, after leader v receives the synchronization message from leader u, leader v sends the timestamp to follower y to begin the Matched algorithm. Once follower y receives the message, it starts executing the following algorithm on the round specified by the timestamp. Followers $x \in C(u)$ are responsible for Matching followers y.

Follower x acts as a matchmaker in this mode. In every round, each follower y_i asks the current neighbour x to be matched with another follower y_j. If multiple followers y_i send a $Matched$ message, follower x matched them in pairs, using their UIDs in ascending order and sends $Matched = \{1, UID\}$ back to each follower y_i where UID is the committee that each follower y_i is matched with. See Figs. 2.7(c),2.7(d). If only one follower y_i sends a $Matched$ message then follower x sends back $Matched = 0, UID$ to inform it that no matches are present. See Figs. 2.7(a). After that follower y_i moves on to the next level of the polylogarithmic tree by activating an edge with the parent of follower x and looks again for a match. See Fig. 2.7(b).

In short, followers y_i might start at different levels of the polylogarithmic tree of $C(u)$. In each round, they activate an edge with the next level until they find a match at their current level. Note that all followers y_i will find a match, since they have a common destination which is the root of committee $C(u)$.

Let us now consider the maximum activated degree of each follower x during the Matched algorithm. In each round, followers y_i might activate an edge with follower x while coming through the lower levels. Each follower x has at most $\log n$ children

Algorithm 5 Matched

```
▷state : round, Matched
▷initial state of node : round = request, Matched = {1, my_UID}
if round == request then
  Send Matched to follower x
if round == receive then
  Receive Matched' = {Match, UID} from follower x
  if Match == 0 then
    Activate edge with parent of x
    Deactivate edge with x
  if Match == 1 then
    myMatch = C(UID)
    round = terminate
if round == terminate then
  Terminate with committee myMatch as its Matched committee
if round == request then
  round = receive
else
  round = request
```

and it is not possible for more than 1 follower y_i to come through each child since, if one child had multiple requests from followers y_i, they would get matched together and terminate as in Fig. 2.7(d). Therefore the maximum activated degree of each follower x can increase by at most $\log n$. Finally note that at this point, the graph of committees consists of directed lines and pairs.

Leader Merging. In this mode, provided that committee $C(v)$ has smaller UID than $C(u)$, leader v activates an edge with leader u by activating edges on the polylogarithmic trees of $C(v)$ and $C(u)$. This process is bound by the diameter of the committees. If we focus on any directed lines in the graph of committees, we can see that we have created a path that consists only of the leaders of the committees in the directed line.

Tree Merging. Every committee leader v executes the asynchronous LineToCompletePolylogarithmicTree algorithm which is the same as the asynchronous LineToCompleteBinaryTree algorithm except that termination criteria requires that your grandfather has $\log n$ children instead of 2 children.

Note here that the whole merging process is finished for this phase. Our final graph consists of a ring graph created by the ring merging process and a collection of wreath graphs with leader u as the root, created by the Leader/Tree merging process. Because of the Tree merging process, there is a polylogarithmic tree consisting of leaders u and v with diameter $O(\frac{\log n}{\log \log n})$. Additionally each leader v is the root of its own polylogarythmic tree with diameter $O(\frac{\log n}{\log \log n})$ from the previous phase. Therefore the diameter of the collection of wreath graphs is $O(\frac{\log n}{\log \log n})$.

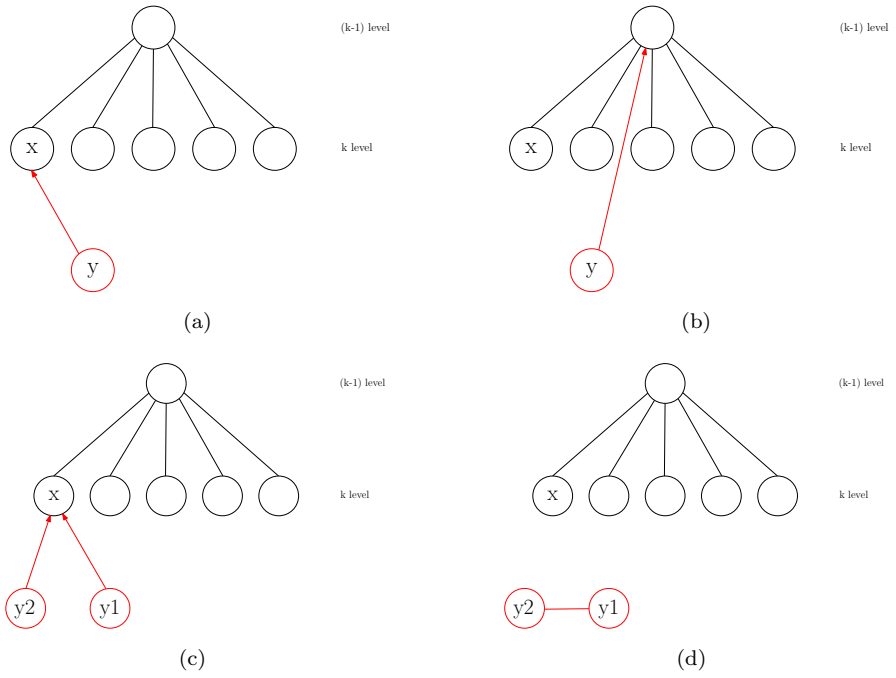

FIGURE 2.7: Figures showing the Matched mode. Figure (a,b) show that committee $C(v)$ goes up one level on the binary tree of committee $C(u)$ if follower y finds no match through follower x. On the other hand, figures (c,d) show that if two committees are in the same round on the same follower x, they get matched together.

2.6.2 GraphToThinWreath Proof

For this algorithm's proof, it is not possible to use the same strategy as the previous algorithms. This is because, while we can prove that this algorithm also requires $O(\log n)$ phases as the previous algorithms, all of our modes require $O(\frac{\log n}{\log \log n})$ but for the tree merging mode which requires $O(\log n)$ and therefore similar analysis would yield $O(\log^2 n)$ running time. Our new strategy is to show that after $O(\log n)$ rounds in which at least one committee is in the tree merging mode in each round, there is only a single committee left in the graph.

Correctness

Lemma 2.16. *Algorithm GraphToThinWreath solves the Depth-*$\frac{\log n}{\log \log n}$ *Tree problem.*

Proof. Since the selection mode of the `GraphToThinWreath` algorithm is identical with the `GraphToWreath` algorithm, we argue that there will be a single committee left in the final graph. This committee consists of a ring subgraph and multiple thinwreath subgraphs. Based on the low level description of the tree merging mode, the diameter of the graph is $O(\frac{\log n}{\log \log n})$. □

Time Complexity

Lemma 2.17. *After $O(\log n)$ tree merging rounds, there is only a single committee left in the graph.*

Proof. We define a tree merging round to be a round in which at least one committee is in the tree merging mode. For the purposes of this proof, we are going to consider each tree merging round to be its own phase. Consider the rounds in which a committee is in the tree merging mode. Observe that in any such round, the leaders of those committees form a forest F, where each committee belongs to a tree of F. Any such tree executes the asynchronous LineToCompletePolylogarithmicTree algorithm. This structure is identical to the structure in the pulling mode of the `GraphToStar` algorithm. The only difference between the pulling mode and the tree merging mode is that they are running different algorithms. But, the asynchronous LineToCompletePolylogarithmicTree and the TreeToStar algorithm have the same running time and both of them merge the trees of committees into single committees. Therefore the two algorithms will have the same number of rounds. Based on Lemma 2.8, there at most $O(\log n)$ phases for the `GraphToStar` algorithm to terminate and every phase includes at most one round of pulling mode and subsequently there are at most $O(\log n)$ rounds of pulling mode. Therefore the `GraphToThinWreath` algorithm can have at most $O(\log n)$ tree merging rounds before a single committee is left in the graph.

□

Lemma 2.18. *The GraphToThinWreath algorithm has $O(\frac{\log^2 n}{\log \log n})$ running time.*

Proof. In order for a committee to enter the tree merging mode, it has to go through some or all of the other modes of the algorithm which have $O(\frac{\log n}{\log \log n})$ running time since all of them are bound by the diameter of the committee. Therefore, for every tree merging round, there can be at most $O(\frac{\log n}{\log \log n})$ rounds from the other modes. Then based on Lemma 2.17, the running time of the algorithm is $O(\frac{\log n}{\log \log n}) \cdot O(\log n) = O(\frac{\log^2 n}{\log \log n})$. □

Edge complexity

Let us consider the maximum possible edges added on each node throughout each phase. Based on the low level description of the modes, each mode adds at most $O(\log n)$ number of edges to each node. From Lemma 2.17, we implicitly know that there can be at most $\log n$ phases until the algorithm terminates. Therefore, the maximum degree of each node is $O(\log^2 n)$. Similarly, since in every round, each node activates at most 1 edge, the maximum edges activated are $O(n \cdot \frac{\log^2 n}{\log \log n})$. Finally, since in every mode, every edge activation is followed by a deactivation, the maximum number of activated edges in $O(n)$.

Theorem 2.19. *For any initial connected graph with polylogarithmic degree, the GraphToThinWreath algorithm solves Depth-$\frac{\log n}{\log\log n}$ Tree in $O(\frac{\log^2 n}{\log\log n})$ time with $O(n\log^2 n)$ total edge activations, $O(n)$ active edges per round and $O(1)$ maximum activated degree.*

2.7 Lower Bounds for the Depth-log n Tree Problem

We will now shift our focus into proving lower bounds for our model. We are going to provide lower bounds for both a centralized model and a distributed one because we want to show that there is an important difference between the two of them.

2.7.1 Centralized Setting

In the centralized setting, everything we have previously defined in the model section stays the same but now every node also has complete knowledge of the graph and a centralized controller can decide what each node will do in each round.

We begin by defining the *potential* of a UID to a node v. The potential describes how far the UID is from node v. We are going to use this definition to measure how fast the identifier can be transmitted throughout the graph.

Definition 2.20. We define the *potential* of a UID_u to v as its minimum "distance" from v. The distance is defined as follows: Consider all nodes w in the network that know UID_u. Compute the length of the shortest path between each node w and node v. The minimum length among all shortest paths is the distance between UID_u and node v. We denote the potential of UID_u to v by $\phi_{u,v}$.

Note that in any initial graph $D = (V, E)$, $\forall u, v \in V, \phi_{u,v} = \max_u \phi_{u,v} \leq n - 1$. Consider any pair of nodes u, v, where $\phi_{u,v} = k$. There are two ways to reduce $\phi_{u,v}$ in each round i:

• **Information Propagation**. Consider all nodes w that currently know UID_u. Compute the shortest path between all pairs of w and v and pick node w that yields the smallest shortest path. Node w can send UID_u to one of its neighbors y that belong to the shortest path between w and v to reduce $\phi_{u,v}$ by 1.

• **Reduce Shortest Paths**. Consider all nodes w that currently know UID_u. Compute the shortest path between all pairs of w and v and pick node w that yields the smallest shortest path with $size = k$. Now consider all pairs of nodes x, y that are potential neighbors and also belong to the shortest path between w and v. Activating xy between one pair of x, y reduces $\phi_{u,v}$ by 1. Activating multiple xy between different pairs in one round can reduce $\phi_{u,v}$ even more but at most by $k/2$.

Observation 1. In order for an algorithm to solve the Depth-log n Tree Problem, $\forall u, v \in V, \phi_{u,v} \leq \log n$.

Lemma 2.21. *Any transformation strategy based on this model requires* $\Omega(\log n)$ *time to solve the Depth-*$\log n$ *tree problem if the initial graph* G_s *is a spanning line.*

Proof. Consider a spanning line where, for simplicity, we call the node that resides at the "left" endpoint of the line u and the node that resides at the "right" endpoint of the line v. According to Observation 1, in order for an algorithm to solve the Depth-$\log n$ tree problem, $\phi_{u,v} \leq \log n$. In the initial graph, $\phi_{u,v} = n - 1$. We know that by using edge activations, we can reduce $\phi_{u,v}$ by half in each round, and by using Information Propagation we can reduce $\phi_{u,v}$ by 1 in each round. Therefore in order for $\phi_{u,v} = \log n$, any algorithm would require at least $\Omega(\log n)$ rounds. □

Lemma 2.22. *Any transformation strategy based on this model that solves the Depth-*$\log n$ *Tree problem in* $O(\log n)$ *time requires* $\Omega(n)$ *edge activations.*

Proof. Let us again consider a spanning line as the initial graph. W.l.o.g. let us assume that the size of the network is odd. Let us call u the node that is the "left" end point of the line and v the "right" endpoint of the line.

Let us assume that in some round i, where $i \leq \log n$, that $\phi_{u,v} \leq \log n$. We can produce the following equation based on the two rules that allow us to reduce the potential: $InitialPotential - \#EdgeActivations - \#MessagesSent \leq \log n$. The maximum value of $MessagesSent$ is $\log n$ and $InitialPotential = n - 1$ and if we add those in the previous equation we get $\#EdgeActivations \geq n-1-2\log n$ and therefore, in order for $\phi_{u,v} \leq \log n$ at least $n - 1 - 2\log n$ edges have to have been activated. □

Lemma 2.23. *Any transformation strategy based on this model that solves the Depth-*$\log n$ *Tree problem in* $O(\log n)$ *time, requires* $\Omega(n/\log n)$ *edge activations per round.*

Proof. From Lemma 2.22 we know that $\phi_{u,v} = 0$ to be possible in $\log n$ time, the following equation has to be true $EdgeActivations = \Omega(n)$. Now, since we are trying to find the minimum number of edge activations per round possible, we can easily do this by dividing the total number of edge activations with the number of rounds. Therefore $EdgeActivationsPerRound \geq \frac{EdgeActivations}{Rounds} \geq \frac{\Omega(n)}{\log n}$. □

Since we have just proven that $\Omega(n)$ edge activations are required in order to solve the Depth-$\log n$ problem given any initial graph, we are now going to prove that $\Theta(n)$ edges are sufficient in order to solve it. First, we are going to informally prove it for the special case of the spanning line graph and afterwards we are going to prove it for general graphs.

Consider a spanning line with nodes $u_1, u_2, \ldots, u_j$ for $j = 1, 2, \ldots, n$. For simplicity, assume that u_1 is the "left" endpoint of the line, u_2 is the neighbor of u_1 etc, u_3 is a neighbor of u_2 etc. In each round i, we activate edge u_j, u_{j+2^i} $\forall$ $\{u_j | (j \mod (2^i) = 1) \wedge (j + 2^i \leq n)\}$. After $\log n$ rounds, the diameter of the shape is equal to $\log n$. Let

us now proceed to analyzing the total edge activations. By definition of the algorithm, in each round i, $\frac{n}{2^i}$ edges are activated. Since the algorithm runs for $\log n$ rounds, we have $\sum_{i=1}^{\log n} \frac{n}{2^i} = n$ total edge activations. We call this algorithm CutInHalf.

Theorem 2.24. *Given any initial graph $D = (V, E)$, the Depth-$\log n$ problem can be solved in $O(\log n)$ time, with $\Theta(n)$ total edge activations.*

Proof. Since we are in a centralized setting, we are first going to perform some global computations that are going to output the specific edges that have to be activated in order for the diameter of the shape to drop to $\log n$. We consider any initial graph $D = (V, E)$ and we pick an arbitrary node called u. First, we compute a spanning tree that starts from node u. Afterwards we compute an Eulerian tour starting from u. This way we can create a virtual ring $D' = (V', E')$ that has $|V'| \leq 2|V|$ and $|E'| \leq 2|E|$. Now in this ring, node u deactivates one of its incident edges and the graph is now a line. We can now execute the CutInHalf algorithm to solve the Depth-$\log n$ Tree problem in $O(\log n)$ time, with $\Theta(n)$ total edge activations. □

2.7.2 Distributed Setting

In this part, we are going to show that there is a difference in the minimum total edge activations required for solving the Depth-$\log n$ problem between the centralized and the distributed model. At this point, we would like to remind to the reader that an algorithm is called *comparison based* if it manipulates the UIDs of the network using comparison operations $(<, >, =)$ only. Our main theorem will show that any deterministic distributed comparison based algorithm requires $\Omega(n \log n)$ total edge activations to solve the Depth-$\log n$ Tree problem in $O(\log n)$ time. Consider two nodes, called u and v , that have received increasing order UIDs that are larger than both u and v during the execution of a deterministic comparison based algorithm. Since nodes are only allowed to compare UIDs between them, nodes u and v the results of the comparison of u and v are exactly the same and thus, u and v must have the same behaviour until they find a different result by comparing receiving UIDS. We are going to use this behaviour to show that any algorithm must activate $\Omega(n \log n)$ total edge activations.

Definition 2.25. Let $U = u_1, u_2, \ldots, u_k$ be a sequence of UIDs of length k. We say that U is an *increasing order sequence* if, for all $i, j, 1 \leq i, j \leq k$, we have $i \leq j$ iff $u_i \leq u_j$.

At this point, we would like to remind to the reader that an algorithm is called *comparison based* if it manipulates the UIDs of the network using comparison operations $(<, >, =)$ only.

Definition 2.26. Let A be a *comparison-based* algorithm executing on an increasing order ring graph. Let i and j be two nodes in the ring graph. We say that i and j are in corresponding states if the UIDs that they both have received from counterclockwise neighbors are a decreasing order sequence and the UIDs they have received are an increasing order sequence and vice versa. Two nodes in corresponding states in round i must have the same behaviour during the execution of an algorithm in round i

Definition 2.27. We define the increasing order ring R as follows. Suppose we have an increasing order sequence U of UIDs to be assigned on a ring with n nodes. We assign the smallest UID from $U = u_1, u_2, \ldots, u_k$ to an arbitrary node and we continue assigning increasing UIDs clockwise (or counterclockwise). We call this an increasing order ring.

Definition 2.28. We define a round of an execution/algorithm to be active if at least one message is sent in it or an edge is activated in it.

Definition 2.29. We define the *k-expo-neighborhood* of node i in ring R of size n, where $0 \leq k \leq n/2$, to consist of the $2 \cdot 2^k + 1$ nodes $i - 2^k, \ldots, i + 2^k$, that is, those that are within distance at most 2^k from node i (including i itself).

Lemma 2.30. *Consider an increasing order ring of size n. Let d_{min} be the initial distance between node d and the node with the minimum UID called d_0. Let d_{max} be the initial distance between node d and the node with me maximum UID called d_{n-1}. Let i and j be two nodes in A, where i_{min}, i_{max} is the minimum distance between i and d_0, d_{max} respectively, and j_{min}, j_{max} is the minimum distance between j and d_0, d_{max} respectively. Let A be a comparison-based algorithm executing in the ring. Then, nodes i and j must be in corresponding states for at least k rounds, where $2^k = \min(\max(i_{min}, i_{max}), \max(j_{min}, j_{max}))$.*

Proof. Note here that nodes i and j are in corresponding states as long as $(((\phi_{d_0,i} > 0) \vee (\phi_{d_{n-1},i} > 0)) \wedge ((\phi_{d_0,j} > 0) \vee (\phi_{d_{n-1},j} > 0))$. In simple terms, i and j are in corresponding states as long as both of them do not know both UID_{d_0} and $UID_{d_{n-1}}$ which follows from definition 2.26. This means that i and j will stop being in corresponding states once one of them learns both d_0 and d_max. By definition, 2^k is the potential between i, j and d_0, d_max and we know that the potential of a UID can only be decreased by information propagation and reducing shortest paths, where information propagation reduces the potential by at most 1 per round and reducing shortest paths reduces it by half. Thus, since the initial potential is 2^k, by applying the potential reduction methods, any algorithm would need at least $k - \log k$ rounds so that $(((\phi_{d_0,i} = 0) \vee (\phi_{d_{n-1},i} = 0)) \wedge ((\phi_{d_0,j} = 0) \vee (\phi_{d_{n-1},j} = 0))$.

□

Observation 2. Any transformation strategy based on this model that solves the Depth-$\log n$ Tree problem in $O(\log n)$ time in an increasing order ring, requires at least $\log n$ active rounds.

Theorem 2.31. *Any deterministic distributed algorithm that solves the Depth-$\log n$ Tree problem in $O(\log n)$ time, requires $\Omega(n \log n)$ total edge activations.*

Proof. Consider an increasing order ring R with n nodes and algorithm A that solves the Depth-$\log n$ problem. Consider the node with the greatest UID in the network, called u_{max}, the node with the smallest UID in the network, called u_1, and the antipodal node of u_{max} called u_c.

First of all, note that in the first round, all nodes except from u_1 and u_{max} are in corresponding states. We can generalize this statement by using Lemma 2.30 to state that in round i, each node whose i-expo-neighborhood does not include both u_1,u_{max} is in a corresponding state with each such node. Therefore those nodes behave the same way e.g. if in round i, one of those c nodes activates an edge, then all c nodes activate an edge. For this proof, we define a round of algorithm A to be *live* if the c nodes activate at least one edge in it, we also define a round of algorithm A to be *asleep* if none of the c nodes activate an edge in it.

We already know that we need at least $\log n$ active rounds to connect u_{max} with u_c from Lemma 2. Our goal here is to prove that $\log n$ of those active rounds also have to be live rounds.

For simplicity, we define the set C where node $u \in C$ if u is in the same corresponding state as u_c (including u_c), the set A where node $u \in A$ if u is not in the same corresponding state as u_c.

Consider an arbitrary round i, where the shortest path between u_{max} and u_c is $|P| = k$. This shortest path can be split into two different paths. The one called P_A that includes nodes $u \in A$ and the one called P_C that includes nodes $v \in C$. Essentially, the potential $\phi_{u_{max},a} \geq |P_C|$ since otherwise, some node $v \in C$ would know $UID_{u_{max}}$ which is impossible by definition of set C. Let us divide our analysis between asleep and live rounds and study how much the potential can be reduced in each round.

• **Asleep rounds.** In each asleep round a, only nodes $u \in A$ can activate edges and $|P_C|$ can only be reduced by at most $l+1$ where l is the total number of live rounds before round a. We can reduce it l by having $u \in A$ activating an edge with each potential neighbor $v \in C$, and reduce it by 1 by having u send $UID_{u_{max}}$ to all $v \in C$.

• **Live rounds.** In each live round l, all nodes can activate an edge so we can reduce $|P_C|$ by $l + 1$ by following the above strategy and additionally, use edge activations between nodes $v \in C$ so that $|P_C|$ is reduced by at most half.

Note here, that Asleep rounds are not enough to reduce the potential to 0 in order to solve the Depth-$\log n$ problem. After $O\log(n)$ asleep rounds, $\phi_{u_{max},a} \geq$

$InitialPotential - (\log n)(l+1) = \frac{n}{2} - (\log n)(l+1)$. Therefore we need at least $\log n$ live rounds to solve the Depth-$\log n$ problem.

We are now examining how many edges are activated in each live round. Before we do that, we list some abbreviations: CN: the number of nodes in the original graph, NRL: number of nodes that were removed in previous live rounds, NRA: number of nodes removed in previous asleep rounds. Recall that in each live round l , at least 1 node $v \in C$ activates an edge and by Lemma 2.30, all nodes $v \in C$ activate an edge. The number of nodes $v \in C$ in round i are $|u| \geq \#CN - NRL - NRA = (n-2) - (\sum_{i=1}^{l-1} 2^i)(\sum_{i=1}^{a} -i(l-1)) - a(l-1)$. The number of edges activated in each round l are at least $|C| \geq |u|$. Therefore the total number of edge activations in live rounds after $\log n$ rounds is at least $(n-2) - (\sum_{i=1}^{\log n} 2^i)(\sum_{i=1}^{\log n} -i(l-1)) - a(l-1) = \Theta(\log n)$. □

2.8 Conclusion

In this Chapter, we investigated a new distributed model involving a network comprised of computing entities that can activate new connections. We defined natural cost measures associated with the edge complexity of actively dynamic algorithms. It turns out that there is a natural trade-off between the time and edge complexity of algorithms. By focusing on the apparently representative task of transforming any initial network from a given family into a target network of (poly)logarithmic diameter, which can then be exploited for global computation or further reconfiguration, we obtained non-trivial insight into this trade-off. At first, we provided the GraphToStar that optimized both time and the maximum activated edges at the cost of having linear degree. We continued in the opposite direction; the GraphToWreath algorithm optimized the maximum degree to a constant at the cost of time. To complete our positive results, we provided a third algorithm, called GrapthToThinWreath that achieves a middle ground between the first two algorithms, for both time and maximum degree. Finally, we accompanied our algorithms with lower bounds for both the centralized and the distributed case of the problem.

There is also a number of technical questions specific to our model and the obtained results. We do not know yet what are the ultimate lower bounds on time for different restrictions on the maximum degree. For maximum degree bounded by a constant our best upper bound is $O(\log^2 n)$ and if bounded by (poly)$\log(n)$ this drops slightly by an $O(\log \log n)$ factor. Can any of these be improved to $O(\log n)$, that is, matching the $\Omega(\log n)$ lower bound on time? It would also be valuable to investigate randomized algorithms for the same problems, like those already developed in overlay networks.

Finally, there are many variants of the proposed model and complexity measures that would make sense and might give rise into further interesting questions and developments. Such variants include anonymous distributed entities which are possibly restricted to treat their neighbors identically even w.r.t. actions (e.g., through local broadcast) and alternative potential neighborhoods, e.g., activating edges at larger distances. Another interesting approach includes introducing a very strict limit to the neighbors of each node in an effort to better mimic real life systems that have a set amount of connections limited by space constraints.

Chapter 3

Growing Graphs

3.1 Introduction

3.1.1 Motivation

Growth processes are found in a variety of networked systems. In nature, crystals grow from an initial nucleation or from a "seed" crystal and a process known as embryogenesis develops sophisticated multicellular organisms, by having the genetic code control tissue growth [54, 55]. In human-made systems, sensor networks are being deployed incrementally to monitor a given geographic area [56, 57], social-network groups expand by connecting with new individuals [58], DNA self-assembly automatically grows molecular shapes and patterns starting from a seed assembly [59–61], and high churn or mobility can cause substantial changes in the size and structure of computer networks [62, 63]. Graph-growth processes are central in some theories of relativistic physics. For example, in dynamical schemes of causal set theory, causets develop from an initial emptiness via a tree-like birth process, represented by dynamic Hasse diagrams [64, 65]. Finally, growth has also been considered in the context of logic and automata research [66, 67]. Though diverse in nature, all these are examples of systems sharing the notion of an underlying graph-growth process. In some, like crystal formation, tissue growth, and sensor deployment, the implicit graph representation is bounded-degree and embedded in Euclidean geometry. In others, like social-networks and causal set theory, the underlying graph might be free from strong geometric constraints but still be subject to other structural properties, as is the special structure of causal relationships between events in casual set theory.

Further classification comes in terms of the source and control of the network dynamics. Sometimes, the dynamics are solely due to the environment in which a system is operating, as is the case in DNA self-assembly, where a pattern grows via random encounters with free molecules in a solution. In other applications, the network dynamics

are, instead, governed by the system. Such dynamics might be determined and controlled by a centralized program or schedule, as is typically done in sensor deployment, or be the result of local independent decisions of the individual entities of the system. Even in the latter case, the entities are often running the same global program, as do the cells of an organism by possessing and translating the same genetic code.

Inspired by such systems, we study a high-level, graph-theoretic abstraction of network-growth processes. We do not impose any strong *a priori* constraints, like geometry, on the graph structure and restrict our attention to *centralized* algorithmic control of the graph dynamics. We do include, however, some weak conditions on the permissible dynamics, necessary for non-triviality of the model and in order to capture realistic abstract dynamics. One such condition is "locality", according to which a newly introduced vertex u' in the neighborhood of a vertex u, can only be connected to vertices within a reasonable distance $d-1$ from u. At the same time, we are interested in growth processes that are "efficient", under meaningful abstract measures of efficiency. We consider two such measures, to be formally defined later, the *time* to grow a given target graph and the number of auxiliary connections, called *excess edges*, employed to assist the growth process. For example, in cellular growth, a useful notion of time is the number of times all existing cells have divided and is usually polylogarithmic in the size of the target tissue or organism. In social networks, it is quite typical that new connections can only be revealed to an individual u' through its connection to another individual u who is already a member of a group. Later, u' can drop its connection to u but still maintain some of its connections to u's group. The dropped connection uu' can be viewed as an excess edge, whose creation and removal has an associated cost, but was nevertheless necessary for the formation of the eventual neighborhood of u'.

The present study is also motivated by work on dynamic graph and network models [68–70]. Research on temporal graphs studies the algorithmic and structural properties of graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, in which $\mathcal{V}$ is a set of time-vertices and $\mathcal{E}$ a set of time-edges of the form (u, t) and (e, t), respectively, t indicating the discrete time at which an instance of vertex u or edge e is available. A substantial part of work in this area has focused on the special case of temporal graphs in which $\mathcal{V}$ is *static*, i.e., time-invariant [30, 71–75]. In overlay networks [5, 51, 76–78] and distributed network reconfiguration [79], $\mathcal{V}$ is a static set of processors that control in a decentralized way the edge dynamics. Even though, in this paper, we do not study our dynamic process from a distributed perspective, it still shares with those models both the fact that dynamics are *active*, i.e., algorithmically controlled, and the locality constraint on the creation of new connections. Nevertheless, our main motivation is theoretical interest. As will become evident, the algorithmic and structural properties of the considered graph-growth process give rise to some intriguing theoretical questions and computationally hard combinatorial optimization problems. Apart from the aforementioned connections to dynamic graph and network

models, we shall reveal interesting similarities to cop-win graphs [80–83]. It should also be mentioned that there are other well-studied models and processes of graph growth, not obviously related to the ones considered here, such as random graph generators [84, 85]. While initiating this study from a centralized and abstract viewpoint, we anticipate that it can inspire work on more applied models, including geometric ones and models in which the growth process is controlled in a decentralized way by the individual network processors. Note that centralized upper bounds can be translated into (possibly inefficient) first distributed solutions, while lower bounds readily apply to the distributed case. There are other studies considering the centralized complexity of problems with natural distributed analogues, as is the work of Scheideler and Setzer on the centralized complexity of transformations for overlay networks [86] and of some of the authors of this paper on geometric transformations for programmable matter [87].

3.1.2 Our Approach

We study the following centralized graph-growth process. The process, starting from a single initial vertex u_0 and applying vertex-generation and edge-modification operations, grows a given target graph G. It operates in discrete time-steps, called slots. In every slot, it generates at most one new vertex u' for every existing vertex u and connects it to u. This is an operation abstractly representing entities that can replicate themselves or that can attract new entities in their local neighborhood or group. Then, for every new vertex u', it connects u' to any (possibly empty) subset of the vertices within a "local" radius around u, described by a distance parameter d, essentially representing that radius plus 1, i.e., as measured from u'. Finally, it removes any (possibly empty) subset of edges whose removal does not disconnect the graph, before moving on to the next slot. These edge-modification operations are essentially capturing, at a high level, the local dynamics present in most of the applications discussed previously. In these applications, new entities typically join a local neighborhood or a group of other entities, which then allows them to easily connect to any of the local entities. Moreover, in most of these systems, existing connections can be easily dropped by a local decision of the two endpoints of that connection. [1]The rest of this paper exclusively focuses on $d = 2$; as formally shown in the appendix, the cases $d = 1$ and $d \geq 3$ admit simple and efficient growth processes.

[1]Despite locality of new connections, a more global effect is still possible. One is for the degree of a vertex u to be unbounded (e.g., grow with the number of vertices). Then u', upon being generated, can connect to an unbounded number of vertices within the "local" radius of u. Another would be to allow the creation of connections between vertices generated in the past, which would enable local neighborhoods to gradually grow unbounded through transitivity relations. In this work, we do allow the former but not the latter. That is, for any edge (u, u') generated in slot t, it must hold that u was generated in some slot $t_{past} < t$ while u' was generated in slot t. Other types of edge dynamics are left for future work.

It is not hard to observe that, without additional considerations, any target graph can be grown by the following straightforward process. In every slot t, the process generates a new vertex u_t which it connects to u_0 and to all neighbors of u_0. The graph grown by this process by the end of slot t, is the clique K_{t+1}, thus, any K_n is grown by it within $n-1$ slots. As a consequence, any target graph G on n vertices can be grown by extending the above process to first grow K_n and then delete from it all edges in $E(K_n) \setminus E(G)$, by the end of the last slot. Such a clique growth process maximizes both complexity parameters that are to be minimized by the developed processes. One is the time to grow a target graph G, to be defined as the number of slots used by the process to grow G, and the other is the total number of deleted edges during the process, called excess edges. The above process always uses $n-1$ slots and may delete up to $\Theta(n^2)$ edges for sparse graphs, such as a path graph or a planar graph.

There is an improvement of the clique process, which connects every new vertex u_t to u_0 and to exactly those neighbors v of u_0 for which vu_t is an edge of the target graph G. At the end, the process deletes those edges incident to u_0 that do not correspond to edges in G, in order to obtain G. If u_0 is chosen to represent the maximum degree, $d_{\max}$, vertex of G, then it is not hard to see that this process uses $n-1-d_{\max}$ excess edges, while the number of slots remains $n-1$ as in the clique process. However, we shall show that there are (poly)logarithmic-time processes using close to linear excess edges for some of those graphs. In general, processes considered *efficient* in this work will be those using (poly)logarithmic slots and linear (or close to linear) excess edges.

The goal of this paper is to investigate the algorithmic and structural properties of such processes of graph growth, with the main focus being on studying the following combinatorial optimization problem, which we call the *Graph Growth Problem*. In this problem, a centralized algorithm is provided with a target graph G, usually from a graph family F, and non-negative integers k and ℓ as its input. The goal is for the algorithm to compute, in the form of a *growth schedule* for G, such a process growing G within at most k slots and using at most ℓ excess edges, if one exists. All algorithms we consider are polynomial-time.[2]

For an illustration of the discussion so far, consider the graph family $F_{star} = \{G \mid G$ is a star on $n = 2^\delta$ vertices$\}$ and assume that edges are activated within local distance $d = 2$. We describe a simple algorithm returning a time-optimal and linear excess-edges growth process, for any target graph $G \in F_{star}$ given as input. To keep this exposition simple, we do not give k and ℓ as input-parameters to the algorithm. The process computed by the algorithm, shall always start from $G_0 = (\{u_0\}, \emptyset)$. In every

[2]Note that this reference to *time* is about the running time of an algorithm computing a growth schedule. But the length of the growth schedule is another representation of time: the time required by the respective growth process to grow a graph. To distinguish between the two notions of time, we will almost exclusively use the term *number of slots* to refer to the length of the growth schedule and *time* to refer to the running time of an algorithm generating the schedule.

slot $t = 1, 2, \ldots, \delta$ and every vertex $u \in V(G_t)$ the process generates a new vertex u', which it connects to u. If $t > 1$ and $u \neq u_0$, it then activates the edge u_0u', which is at distance 2, and removes the edge uu'. It is easy to see that by the end of slot t, the graph grown by this process is a star on 2^t vertices centered at u_0 (see Figure 3.1) . Thus, the process grows the target star graph G within $\delta = \log n$ slots. By observing that $2^t/2 - 1$ edges are removed in every slot t, it follows that a total of $\sum_{1 \leq t \leq \log n} 2^{t-1} - 1 < \sum_{1 \leq t \leq \log n} 2^t = O(n)$ excess edges are used by the process. Note that this algorithm can be easily designed to compute and return the above growth schedule for any $G \in F_{star}$ in time polynomial in the size $|\langle G \rangle|$ of any reasonable representation of G.

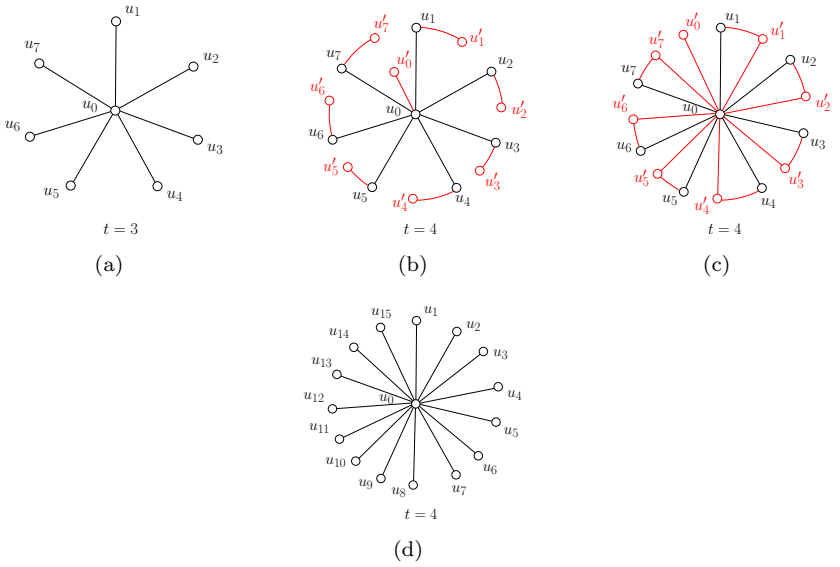

FIGURE 3.1: The operations of the star graph process in slot $t = 4$. (a) A star of size 2^3 grown by the end of slot 3. (b) For every u_i, a vertex u'_i is generated by the process and is connected to u_i. (c) New vertices u'_i are connected to u_0. (d) Edges between peripheral-vertices are being removed to obtain the star of size 2^4 grown by the end of slot 4. Here, we also rename the vertices for clarity.

Note that there is a natural trade-off between the number of slots and the number of excess edges that are required to grow a target graph. That is, if we aim to minimize the number of slots (resp. of excess edges) then the number of excess edges (resp. slots) increases. To gain some insight into this trade-off, consider the example of a path graph G on n vertices $u_0, u_1, ..., u_{n-1}$, where n is even for simplicity. If we are not allowed to activate any excess edges, then the only way to grow G is to always extend the current path from its endpoints, which implies that a schedule that grows G must have at least $\frac{n}{2}$ slots. Conversely, if the growth schedule has to finish after $\log n$ slots, then G can only be grown by activating $\Omega(n)$ excess edges.

In this paper, we mainly focus on this trade-off between the number of slots and the number of excess edgesthat are needed to grow a specific target graph G. In general, given a growth schedule σ, any excess edge can be removed just after the last time it is used as a "relay" for the activation of another edge. In light of this, an algorithm computing a growth schedule can spend linear additional time to optimize the slots at which excess edges are being removed. A complexity measure capturing this is the *maximum excess edges lifetime*, defined as the maximum number of slots for which an excess edge remains active. Our algorithms will generally be aiming to minimize this measure. When the focus is more on the trade-off between the slots and the number of excess edges, we might be assuming that all excess edges are being removed in the last slot of the schedule, as the exact timing of deletion makes no difference w.r.t. these two measures.

3.1.3 Contribution

Section 2 presents the model and problem statement and gives two basic subprocesses that are recurrent in our growth processes. In Section 3.2.4, we provide some basic propositions that are crucial to understanding the limitations on the number of slots and the number of excess edges required for a growth schedule of a graph G. We then use these propositions to provide some lower bounds on the number of slots.

In Section 3.3, we study the *zero-excess growth schedule* problem, where the goal is to decide whether a graph G has a growth schedule of k slots and $\ell = 0$ excess edges. We define the *candidate elimination ordering* of a graph G as an ordering $v_1, v_2, \ldots, v_n$ of $V(G)$ so that for every vertex v_i, there is some v_j, where $j < i$ such that $N[v_i] \subseteq N[v_j]$ in the subgraph induced by $v_i, \ldots, v_n$, for $1 \leq i \leq n$. We show that a graph has a growth schedule of $k = n - 1$ slots and $\ell = 0$ excess edges if and only if it is has a *candidate elimination ordering*. Our main positive result is a polynomial-time algorithm that computes whether a graph has a growth schedule of $k = \log n$ slots and $\ell = 0$ excess edges. If it does, the algorithm also outputs such a growth schedule. On the negative side, we give two strong hardness results. We first show that the decision version of the zero-excess growth schedule problem is NP-complete. Then, we prove that, for every $\varepsilon > 0$, there is no polynomial-time algorithm which computes a $n^{\frac{1}{3}-\varepsilon}$-approximate zero-excess growth schedule, unless P = NP.

In Section 3.4, we study growth schedules of (poly)logarithmic slots. We provide two polynomial-time algorithms. One outputs, for any tree graph, a growth schedule of $O(\log^2 n)$ slots and only $O(n)$ excess edges, and the other outputs, for any planar graph, a growth schedule of $O(\log n)$ slots and $O(n \log n)$ excess edges. Finally, we give lower bounds on the number of excess edges required to grow a graph, when the number of slots is fixed to $\log n$.

In the conclusion, we also discuss some interesting problems opened by this work.

3.2 Preliminaries

3.2.1 Model and Problem Statement

A *growing graph* is modeled as an undirected dynamic graph $G_t = (V_t, E_t)$, where $t = 1, 2, \ldots, k$ is a discrete time-step, called *slot*. The dynamics of G_t are determined by a centralized *growth process* (also called *growth schedule*) σ, defined as follows. The process always starts from the initial graph instance $G_0 = (\{u_0\}, \emptyset)$, containing a single initial vertex u_0, called the *initiator*. In every slot t, the process updates the current graph instance G_{t-1} to generate the next, G_t, according to the following vertex and edge update rules. The process first sets $G_t = G_{t-1}$. Then, for every $u \in V_{t-1}$, it adds at most one new vertex u' to V_t (*vertex generation* operation) and adds to E_t the edge uu' alongside any subset of the edges $\{vu' \mid v \in V_{t-1}$ is at distance at most $d-1$ from u in $G_{t-1}\}$, for some integer *edge-activation distance* $d \geq 1$ fixed in advance (*edge activation* operation). Throughout the rest of the paper, $d = 2$ is always assumed (other edge-activation distances are being studied in the appendix). We call u' the vertex generated by the process for vertex u in slot t. We also say that u is the *parent* of u' and that u' is the *child* of u at slot t and write $u \xrightarrow{t} u'$. The process completes slot t after deleting any (possibly empty) subset of edges from E_t (*edge deletion* operation). We also denote by V_t^+, E_t^+, and E_t^- the set of vertices generated, edges activated, and edges deleted in slot t, respectively. Then, $G_t = (V_t, E_t)$ is also given by $V_t = V_{t-1} \cup V_t^+$ and $E_t = (E_{t-1} \cup E_t^+) \setminus E_t^-$. Deleted edges are called *excess edges* and we restrict attention to excess edges whose deletion does not disconnect G_t. We call G_t the graph *grown* by process σ after t slots and call the final instance, G_k, the *target graph* grown by σ. We also say that σ is a *growth schedule for* G_k that grows G_k in k slots using ℓ *excess edges*, where $\ell = \sum_{t=1}^{k} |E_t^-|$, i.e., ℓ is equal to the total number of deleted edges. This brings us to the main problem studied in this paper:

Graph Growth Problem: Given a target graph G and non-negative integers k and ℓ, compute a growth schedule for G of at most k slots and at most ℓ excess edges, if one exists.

The *target graph* G, which is part of the input, will often be drawn from a given graph family F, e.g., the family of planar graphs. Throughout, n denotes the number of vertices of the target graph G. In this paper, computation is always to be performed by a *centralized* polynomial-time algorithm.

Let w be a vertex generated in a slot t, for $1 \leq t \leq k$. The *birth path* of vertex w is the unique sequence $B_w = (u_0, u_{i_1}, \ldots, u_{i_{p-1}}, u_{i_p} = w)$ of vertices, where $i_p = t$ and $u_{i_{j-1}} \xrightarrow{i_j} u_{i_j}$, for every $j = 1, 2, \ldots, p$. That is, B_w is the sequence of vertex generations

that led to the generation of vertex w. Furthermore, the *progeny* of a vertex u is the set P_u of descendants of u, i.e., P_u contains those vertices v for which $u \in B_v$ holds.

In what follows, we give a formal and detailed description of a graph growth schedule.

Definition 3.1 (growth schedule for $d = 2$)**.** Let $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k, \mathcal{E})$ be an ordered sequence of sets, where $\mathcal{E}$ is a set of edges, and each $\mathcal{S}_i = \{(u_1, v_1, E_1), (u_2, v_2, E_2), \ldots, (u_q, v_q, E_q)\}$ is an unordered set of ordered tuples $\{(u_j, v_j, E_j) : 1 \leq j \leq q\}$ such that, for every j, u_j and v_j are vertices (where u_j gives birth to v_j) and E_j is a set of edges incident to v_j such that $u_j v_j \in E_j$. Suppose that, for every $2 \leq i \leq k$, the following conditions are all satisfied:

- each of the sets $\{v_1, v_2, \ldots, v_q\}$ and $\{u_1, u_2, \ldots, u_q\}$ contain q distinct vertices,
- each vertex $v_j \in \{v_1, v_2, \ldots, v_q\}$ does not appear in any set among $\mathcal{S}_1, \ldots, \mathcal{S}_{i-1}$ (i.e., v_j is "born" at slot i),
- for each vertex $u_j \in \{u_1, u_2, \ldots, u_q\}$, there exists exactly one set among $\mathcal{S}_1, \ldots, \mathcal{S}_{i-1}$ which contains a tuple (u', u_j, E') (i.e., u_j was "born" at a slot before slot i).

Let $i \in \{2, \ldots, k\}$, and let u be a vertex that has been generated at some *slot* $i' \leq i$, that is, u appears in at least one tuple of a set among $\mathcal{S}_1, \ldots, \mathcal{S}_i$. We denote by E^i the union of all edge sets that appear in the tuples of the sets $\mathcal{S}_1, \ldots, \mathcal{S}_i$; E^i is the set of all edges activated until slot i. We denote by $N_i(u)$ the set of neighbors of u in the set E^i. If, in addition, $\mathcal{E} \subseteq E^k$ and, for every $2 \leq i \leq k$ and for every tuple $(u_j, v_j, E_j) \in \mathcal{S}_i$, we have that $N_i(v_j) \subseteq N_i(u_j)$, then σ is a *growth schedule* for the graph $G = (V, E^k \setminus \mathcal{E})$, where V is the set of all vertices which appear in at least one tuple in σ. The number k of sets in σ is the *length* of σ. Finally, we say that G is *constructed* by σ with k slots and with $|\mathcal{E}|$ excess edges.

3.2.2 Basic Subprocesses

We start by presenting simple algorithms for two basic growth processes that are recurrent both in our positive and negative results. One is the process of growing any path graph and the other is that of growing any star graph. Both returned growth schedules use a number of slots which is logarithmic and a number of excess edges which is linear in the size of the target graph. Logarithmic being a trivial lower bound on the number of slots required to grow graphs of n vertices, both schedules are optimal w.r.t. their number of slots. As will shall later follow from Corollary 3.35 in Section 3.4.3, they are also optimal w.r.t. the number of excess edges used for this time-bound.

`Path` algorithm: Let u_0 always be the "left" endpoint of the path graph being grown. For any target path graph G on n vertices, the algorithm computes a growth schedule

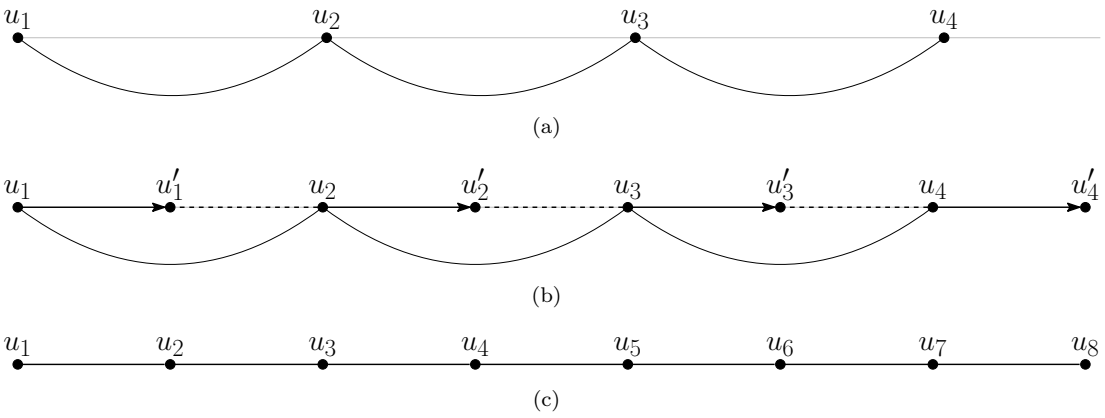

FIGURE 3.2: (a) The path graph at the beginning of slot 3. (b) Vertex generation and edge activation step (steps 1 and 2). The arrows represent vertex generations, while dotted lines represent the edges added to vertices of distance 2. (c) Third slot of the ***path*** algorithm.

for G as follows. For every slot $1 \leq t \leq \lceil \log n \rceil$ and every vertex $u_i \in V_{t-1}$, for $0 \leq i \leq 2^{t-1} - 1$, it generates a new vertex u'_i and connects it to u_i. Then, for all $0 \leq i \leq 2^{t-1} - 2$, it connects u'_i to u_{i+1} and deletes the edge $u_i u_{i+1}$. Finally, it renames the vertices $u_0, u_1, \ldots, u_{2^t-1}$ from left to right, before moving on to the next slot.

Figure 3.2 shows an example slot produced by the path algorithm. The pseudo-code of the algorithm can be found in 6. Note that the pseudo-code growth schedule of Algorithm 6 reserves every edge deletion operation until the last slot.

Lemma 3.2. *For any path graph G on n vertices, the* `path` *algorithm computes in polynomial time a growth schedule σ for G of $\lceil \log n \rceil$ slots and $O(n)$ excess edges.*

Proof. It is easy to see by the description and by Figure 3.2 that the graph constructed is a path subgraph on n vertices. To expand on this, in every slot, we maintain a path graph but we double its size. This process requires $\lceil \log n \rceil$ slots by design since in every slot, for every vertex the process generates a new vertex (apart from the last slot) and after $\lceil \log n \rceil$ slots, the size of the current graph will be n. For the excess edges, consider that in the whole process at the end of every slot t, every edge activated in the previous slot $t-1$ is deleted. Every edge activated in the process apart from those in the last slot is an excess edge. For every vertex generation there are at most 2 edge activations that occur in the same slot and there are $n-1$ total vertex generations in total which means that the total edge activations are $2(n-1)$. Therefore, the excess edges are at most $2(n-1) - (n-1) = O(n)$ since the final path graph has $n-1$ edges. Finally, note that if an excess edge is activated in slot t, then it is deleted in slot $t+1$ which results in maximum lifetime of 1. □

Star **algorithm:** The description of the algorithm can be found in Section 3.1.2.

Algorithm 6 Path growth schedule.

Input: A path graph $G = (V, E)$ on n vertices.
Output: A growth schedule for G.

$V = u_1$
for $k = 1, 2, \ldots, \lceil \log n \rceil$ **do**
  $\mathcal{S}_k = \emptyset; \quad V_k = \emptyset; \quad \mathcal{E} = \emptyset$
  for each vertex $u_i \in V_k$ **do**
    $\mu(k) = i + \lceil n/2^k \rceil$
    if $u_{\mu(t)} \in V$ **then**
      $\mathcal{S}_k = \mathcal{S}_k \cup \{(u_i, u_{\mu(k)}, \{u_i u_{\mu(k)}, (u_{\mu(k)} u_{\mu(k-1)} : u_{\mu(k-1)} \in V)\})\}$
      if $k < \lceil \log n \rceil$ **then**
        $\mathcal{E} = \mathcal{E} \cup \{u_i u_{\mu(k)}, (u_{\mu(k)} u_{\mu(k-1)} : u_{\mu(k-1)} \in V)\})\}$
      $V_k \leftarrow V_k \cup u_{\mu(k)}$
  $V \leftarrow V \cup V_k$
return $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{\lceil \log n \rceil}, \mathcal{E})$

Lemma 3.3. *For any star graph G on n vertices, the* `star` *algorithm computes in polynomial time a growth schedule σ for G of $\lceil \log n \rceil$ slots and $O(n)$ excess edges.*

Proof. Let u_0 be the initiator and n the size of the star graph. By construction, we can see that in every slot a star graph is maintained. In order to terminate with a star graph of size n, we require $\lceil \log n \rceil$ slots since for every vertex a new vertex is generated in each slot and therefore after $\lceil \log n \rceil$ slots, the graph will have size n.

For every vertex generation, there are at most two edge activations. Since there are $n - 1$ vertices generated in total, there are $2(n - 1)$ total edge activations. Therefore, the excess edges are at most $2(n - 1) - (n - 1) = O(n)$. Finally, note that if an excess edge is activated in slot t, then it is deleted in slot $t + 1$ which results in maximum lifetime of 1. □

3.2.3 The Case $d = 1$ and $d \geq 3$

In this section, we show that for edge-activation distance $d = 1$ or $d \geq 3$ there are simple but very efficient algorithms for finding growth schedules. As a warm-up, we begin with a simple observation for the special case where the edge-activation distance d is equal to 1.

Observation 3. For $d = 1$, every graph G that has a growth schedule is a tree graph.

Proposition 3.4. *For $d = 1$, the shortest growth schedule σ of a path graph (resp. a star graph) on n vertices has $\lceil n/2 \rceil$ (resp. $n - 1$) slots.*

Proof. Let G be the path graph on n vertices. By definition of the model, increasing the size of the path can only be achieved by generating one new vertex at each of the endpoints of the path. The size of a path can only be increased by at most 2 in each

slot, where for each endpoint of the path a new vertex that becomes the new endpoint of the path is generated. Therefore, in order to create any path graph of size n would require at least $\lceil n/2 \rceil$ slots. The growth schedule where one vertex is generated at each of the endpoints of the path in each slot creates the path graph of n vertices in $\lceil n/2 \rceil$ slots.

Now let G be the star graph with $n-1$ leaves. Increasing the size of the star graph can only be achieved by generating new leaves directly connected to the center vertex, and this can occur at most once per slot. Therefore, the growth schedule of G requires exactly $n-1$ slots. □

Observation 4. Let $d = 1$ and $G = (V, E)$ be a tree graph with diameter $diam$. Then any growth schedule σ that constructs G requires at least $\lceil diam/2 \rceil$ slots.

Proof. Consider a path p of size $diam$ that realizes the diameter of graph G. By Proposition 3.4 we know that p alone requires a growth schedule with at least $\lceil diam/2 \rceil$ slots. □

Observation 5. Let $d = 1$ and $G = (V, E)$ be a tree graph with maximum degree $\Delta(G)$. Then any growth schedule σ that constructs G requires at least $\Delta(G)$ slots.

Proof. Consider a vertex $u \in G$ with degree $\Delta(G)$ and let $G' = (V', E')$ be a subgraph of G, such that $V' = N_G[u]$ and $E' = E(N_G[u])$. Notice that G' is a star graph of size $\Delta(G)$. By Proposition 3.4, we know that the growth schedule of G' alone has at least $\Delta(G)$ slots. □

We now provide an algorithm, called ***trimming***, that optimally solves the graph growth problem for $d = 1$. We begin with the following simple observation.

Observation 6. Let $d = 1$. Consider a tree graph G and a growth schedule σ for it. Denote by G_t the graph grown so far until the end of slot t of σ. Then any vertex generated in slot t must be a leaf vertex in G_t.

Proof. Every vertex u generated in slot t has degree equal to 1 at the end of slot t by definition of the model for $d = 1$. Therefore every vertex u in graph G_t must be a leaf vertex. □

The ***trimming*** algorithm, see Algorithm 7 follows a bottom-up approach for building the intended growth schedule $\sigma = (\mathcal{S}_k, \mathcal{S}_{k-1}, \ldots \mathcal{S}_1, \mathcal{E})$, where $\mathcal{E} = \emptyset$. At every iteration $i = 1, 2, \ldots$ of the algorithm, we consider the leaves of the current tree graph and we create the parent-child pairs of the currently last slot $\mathcal{S}_{k+1-i}$ of the schedule. Then we remove from the current tree graph all the leaves that were included in a parent-child pair at this iteration of the algorithm, and we recurse. The process is repeated until

Algorithm 7 Trimming Algorithm, where $d = 1$.

Input: A target tree graph $G = (V, E)$ on n vertices.
Output: An optimal growth schedule for G.

$k \leftarrow 1$
while $V \neq \emptyset$ **do**
  for each leaf vertex $v \in V$ and its unique neighbor $u \in V$ **do**
    if u is not marked as a "parent in S_k" **then**
      Mark u as a "parent in $\mathcal{S}_k$"
      $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup \{(u, v, \{uv\})\}$
      $V \leftarrow V \setminus \{v\}$
  $\mathcal{S}_{k+1} = \emptyset$; $k \leftarrow k + 1$
return $\sigma = (\mathcal{S}_k, \mathcal{S}_{k-1}, \ldots \mathcal{S}_1, \emptyset)$

graph G has a single vertex left which is added in the first slot of σ as the initiator. In the next theorem we show that the algorithm produces an optimum growth schedule.

Theorem 3.5. *For $d = 1$, the* `trimming` *algorithm computes in polynomial time an optimum (shortest) growth schedule σ of κ slots for any tree graph G.*

Proof. Let $\sigma = (\mathcal{S}_1, \ldots, \mathcal{S}_k, \emptyset)$ be the growth schedule obtained by the ***trimming*** algorithm (with input G). Suppose that σ is not optimum, and let $\sigma' \neq \sigma$ be an optimum growth schedule for G. That is, $\sigma = (\mathcal{S}'_1, \ldots, \mathcal{S}'_{k'}, \emptyset)$, where $k' < k$. Denote by $(L_1, L_2, \ldots L_k)$ and $(L'_1, L'_2, \ldots L'_{k'})$ the sets of vertices generated in each slot of the growth schedules σ and σ', respectively. Note that $\sum_{i=1}^{k} |L_i| = \sum_{i=1}^{k'} |L'_i| = n - 1$. Among all optimum growth schedules for G, we can assume without loss of generality that σ' is chosen such that the vector $(|L'_{k'}|, |L'_{k'-1}|, \ldots |L'_1|)$ is lexicographically largest.

Recall that the ***trimming*** algorithm builds the growth schedule σ backwards, i.e., it first computes $\mathcal{S}_k$, it then computes $\mathcal{S}_{k-1}$ etc. At the first iteration, the ***trimming*** algorithm collects all vertices which are parents of at least one leaf in the tree and for each of them will generate a new vertex in $\mathcal{S}_k$. Then, the algorithm removes all leaves that are generated in $\mathcal{S}_k$, and it recursively proceeds with the remaining tree after removing these leaves.

Let ℓ be the number of slots such that the growth schedules σ and σ' generate the same number of leaves in their last ℓ slots, i.e., $|L_{k-i}| = |L'_{k'-1}|$, for every $i \in \{0, 1, \ldots, \ell - 1\}$, but $|L_{k-\ell}| \neq |L'_{k'-\ell}|$. Suppose that $\ell \leq k - 1$. Note by construction of the ***trimming*** algorithm that, since $|L_k| = |L'_{k'}|$, both growth schedules σ and σ' generate exactly one leaf for each vertex which is a parent of a leaf in G. That is, in their last slot, both σ and σ' have the same parents of new vertices; they might only differ in which leaves are generated for these parents. Consider now the graph G_{k-1} (resp. $G'_{k'-1}$) that is obtained by removing from G the leafs of L_k (resp. of $L'_{k'}$). Then note that G_{k-1} and $G'_{k'-1}$ are isomorphic. Similarly it follows that, if we proceed removing from the current graph the vertices generated in the last ℓ slots of

the schedules σ and σ', we end up with two isomorphic graphs $G_{k-\ell+1}$ and $G'_{k'-\ell+1}$. Recall now that, by our assumption, $|L_{k-\ell}| \neq |L'_{k'-\ell}|$. Therefore, since the ***trimming*** algorithm always considers all possible vertices in the current graph which are parents of a leaf (to give birth to a leaf in the current graph), it follows that $|L_{k-\ell}| > |L'_{k'-\ell}|$. That is, at this slot the schedule σ' misses at least one potential parent u of a leaf v in the current graph $G'_{k'-\ell+1}$. This means that the tuple $(u, v, \{uv\})$ appears at some other slot $\mathcal{S}'_j$ of σ', where $j < k' - \ell$. Now, we can move this tuple from slot $\mathcal{S}'_j$ to slot $\mathcal{S}'_{k'-\ell}$, thus obtaining a lexicographically largest optimum growth schedule than σ', which is a contradiction.

Therefore $\ell \geq k$, and thus $\ell = k$, since $\sum_{i=1}^{k} |L_i| = \sum_{i=1}^{k'} |L'_i| = n - 1$. This means that σ and σ' have the same length. That is, σ is an optimum growth schedule. □

We move on to the case of $d \geq 4$, and we show that for any graph G, there is a simple algorithm that computes a growth schedule of an optimum number of slots and only linear number of excess edges in relation to the size of the graph.

Lemma 3.6. *For $d \geq 4$, any given graph $G = (V, E)$ on n vertices can be grown with a growth schedule σ of $\lceil \log n \rceil$ slots and $O(n)$ excess edges.*

Proof. Let $G = (V, E)$ be the target graph, and $G_t = (V_t, E_t)$ be the grown graph at the end of slot t. When the growth schedule generates a vertex w, w is matched with an unmatched vertex of the target graph G. For any pair of vertices $v, w \in G_{\lceil \log n \rceil}$ that have been matched with a pair of vertices $v_j, w_j \in G$, respectively, if $(v_j, w_j) \in E$, then $(v, w) \in E_{\lceil \log n \rceil}$, and if $(v_j, w_j) \notin E$, then $(v, w) \notin E_{\lceil \log n \rceil}$.

To achieve growth of G in $\lceil \log n \rceil$ slots, for each vertex of G_t the process must generate a new vertex at slot $t + 1$, except possibly for the last slot of the growth schedule. To prove the lemma, we show that the growth schedule maintains a star as a spanning subgraph of G_t, for any $t \leq \lceil \log n \rceil$, with the initiator u as the center of the star. Trivially, the children of u belong to the star, provided that the edge between them is not deleted until slot $\lceil \log n \rceil$. The children of all leaves of the star are at distance 2 from u, therefore the edge between them and u are activated at the time of their birth.

The above schedule shows that the distance of any two vertices is always less or equal to four. Therefore, for each vertex w that is generated in slot t and is matched to a vertex $w_j \in G$, the process activates the edges with each vertex u that has been generated and matched to vertex $u_j \in G_j$ and $(w_j, u_j) \in E$. Finally, the number of the excess edges that we activate are at most $2n - 1$ (i.e., the edges of the star and the edges between parent and child vertices). Any other edge is activated only if it exists in G. □

It is not hard to see that the proof of Lemma 3.6 can be slightly adapted such that, instead of maintaining a star, we maintain a clique. The only difference is that, in this case, the number of excess edges increases to at most $O(n^2)$ (instead of at most $O(n)$). On the other hand, this method of always maintaining a clique has the benefit that it works for $d = 3$, as the next lemma states.

Lemma 3.7. *For $d \geq 3$, any given graph $G = (V, E)$ on n vertices can be grown with a growth schedule σ of $\lceil \log n \rceil$ slots and $O(n^2)$ excess edges.*

3.2.4 Basic Properties on $d = 2$

For the rest of the paper, we always assume that $d = 2$. In this section, we show some basic properties for growing a graph G which restrict the possible growth schedules and we also provide some lower bounds on the number of slots. In the next proposition we show that the vertices generated in each slot form an independent set in the grown graph, i.e., any pair of vertices generated in the same slot cannot have an edge between them in the target graph.

Proposition 3.8. *The vertices generated in a slot form an independent set in the target graph G.*

Proof. Let G_{t-1} be our graph at the beginning of slot t. Consider any pair of vertices u_1, u_2 that have minimal distance between them, in other words, they are neighbors and $dist = 1$. Assume that for vertices u_1, u_2 new vertices v_1, v_2 are generated in slot t, respectively. The distance between vertices v_1, v_2 in slot t just after they are generated is $dist = 3$ and therefore the process cannot activate an edge between them. Finally, for any other pair of non-neighboring vertices, the distance between their children is $dist > 3$, thus remaining an independent set. □

Proposition 3.9. *Consider any growth schedule σ for graph G. Let t_1, t_2, $t_1 \leq t_2$, be the slots in which a pair of vertices u, w is generated, respectively. Let $dist_{t_2}$ be the distance between u and w at the end of slot t_2. Then, at the end of any slot $t \geq t_2$, $dist_t \geq dist_{t_2}$.*

Proof. Given that the $d = 2$, for any vertex that is generated at slot t, edges can only be activated with its parent and with the neighbors of its parent.

Let u be a vertex that is generated for a vertex u' at t_1, and w be a vertex that is generated for a vertex w' at t_2. Let G_{t_2-1} be the graph at the beginning of slot t_2, and P, $|P| = d$, be the shortest path between u and w in G_{t_2-1}. We distinguish two cases:

1. For vertex u and/or w new vertices that are connected to all neighbors of u and/or w are generated. In this case, the path that contains the new vertices will clearly be larger than d.

2. In this case, for some vertex p in the path between u and w a new vertex p' is generated and all its edges with the neighbors of p are activated. In this case, the path that passes through p' will clearly have the same size as P.

It is then obvious that no growth schedule starting from G_{t_2-1} can reduce the shortest distance between u and w. □

Proposition 3.10. *Consider t_1, t_2, where $t_1 \leq t_2$, to be the slots in which a pair of vertices u, w is generated by a growth schedule σ for graph G, respectively, and edge (u, w) is not activated at t_2. Then any pair of vertices v, z cannot be neighbors in G if u belongs to the birth path of v and w belongs to the birth path of z.*

Proof. Given that the edge between vertices u and w is not activated, and by Proposition 3.9, the children of u will always have distance at least 2 from w (i.e., edges of these children can only be activated with the vertices that belong to the neighborhood of their parent vertex, and no edge activations can reduce their distance). Sequentially, the same holds also for the children of w. All vertices that belong to the progeny P_u of u (i.e., each vertex z such that $u \in B_z$) have to be in distance at least 2 from w, therefore they cannot be neighbors with any vertex in P_w. □

We will now provide some lower bounds on the number of slots for any growth schedule σ for graph G.

Lemma 3.11. *Assume that graph G has chromatic number $\chi(G)$. Then any growth schedule σ that grows G requires at least $\chi(G)$ slots.*

Proof. Assume that there exists a growth schedule σ that can grow graph G in $k < \chi(G)$ slots. By Proposition 3.8, the vertices generated in each slot t_i for $i = 1, 2, ..., k$ must form an independent set in G. Therefore, we could color graph G using k colors which contradicts the original statement that $\chi(G) > k$. □

Lemma 3.12. *Assume that graph G has clique number $\omega(G)$. Then any growth schedule σ for G requires at least $\omega(G)$ slots.*

Proof. By Proposition 3.8, we know that every slot must contain an independent set of the graph and it cannot contain more than one vertex from clique q. By the pigeon hole principle, it follows that σ must have at least c slots. □

3.3 Growth Schedules of Zero Excess Edges

In this section, we study which target graphs G can be grown using $\ell = 0$ excess edges for $d = 2$. We begin by providing an algorithm that decides whether a graph G can be grown by any schedule σ. We build on to that, by providing an algorithm that computes

a schedule of $k = \log n$ slots for a target graph G, if one exists. We finish with our main technical result showing that computing the smallest schedule for a graph G is NP-complete and any approximation of the shortest schedule cannot be within a factor of $n^{\frac{1}{3}-\varepsilon}$ of the optimal solution, for any $\varepsilon > 0$, unless $P = NP$. First, we check whether a graph G has a growth schedule of $\ell = 0$ excess edges. Observe that a graph G has a growth schedule if and only if it has a schedule of $k = n - 1$ slots.

Definition 3.13. Let $G = (V, E)$ be any graph. A vertex $v \in V$ can be the last generated vertex in a growth schedule σ of $\ell = 0$ for G if there exists a vertex $w \in V \backslash \{v\}$ such that $N[v] \subseteq N[w]$. In this case, v is called a *candidate* vertex and w is called the *candidate parent* of v. Furthermore, the set of candidate vertices in G is denoted by $S_G = \{v \in V : N[v] \subseteq N[w] \text{ for some } w \in V \setminus \{v\}\}$, see Figure 3.3.

Definition 3.14. A candidate elimination ordering of a graph G is an ordering $v_1, v_2, \ldots, v_n$ of $V(G)$ such that v_i is a candidate vertex in the subgraph induced by $v_i, \ldots, v_n$, for $1 \leq i \leq n$.

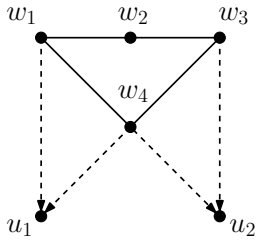

FIGURE 3.3: Consider the above graph G_t to be the graph grown after slot t. Vertices u_1 and u_2 are candidate vertices. The arrows represent all possible vertex generations in the previous slot t. Vertices w_1 and w_4 are candidate parents of u_1, while w_3 and w_4 are candidate parents of u_2.

Lemma 3.15. *A graph G has a growth schedule of $n - 1$ slots and $\ell = 0$ excess edges if and only if G has a candidate elimination ordering.*

Proof. By definition of the model, whenever a vertex u is generated for a vertex w in a slot t, only edges between u and vertices in $N[w]$ can be activated, which means that $N[u] \subseteq N[w]$. Since $\ell = 0$, this property stays true in G_{t+1}. Therefore, any vertex u generated in slot t, is a candidate vertex in graph G_{t+1}. □

The following algorithm can decide whether a graph has a candidate elimination ordering, and therefore, whether it can be grown with a schedule of $n - 1$ slots and $\ell = 0$ excess edges. The algorithm computes the slots of the schedule in reverse order.

Candidate elimination ordering **algorithm**: Given the graph $G = (V, E)$, the algorithm finds all candidate vertices and deletes an arbitrary candidate vertex and its

Algorithm 8 Candidate elimination order

Input: A graph $G = (V, E)$ on n vertices.
Output: A growth schedule for G, if it exists

```
1: for k = n − 1 downto 1 do
2:   S_k = ∅
3:   for every vertex v ∈ V do
4:     if (N[v] ⊆ N[u], for some vertex u ∈ V \ {v}) ∧ (S_k = ∅) then {v is a new
       candidate vertex}
5:       S_k ← {(u, v, {vw : w ∈ N(v)})}
6:       V ← V \ {v}
7:   if S_k = ∅ then
8:     return "NO"
9: return σ = (S_1, S_2, ..., S_{n−1}, ∅)
```

incident edges. The deleted vertex is added in the last empty slot of the schedule σ. The algorithm repeats the above process until there is only a single vertex left. If that is the case, the algorithm produces a growth schedule. If the algorithm cannot find any candidate vertex for removal, it decides that the graph cannot be grown.

Lemma 3.16 ($\star$)**.** *Let $v \in S_G$. Then G has a candidate elimination ordering if and only if $G - v$ has a candidate elimination ordering.*

Proof. Let c be a candidate elimination ordering of $G - v$. Then, generating vertex v at the end of c trivially results in a candidate elimination ordering of G.

Conversely, let c be a candidate elimination ordering of G. If v is the last vertex in c, then $c \setminus \{v\}$ is trivially a candidate elimination ordering of $G - v$. Suppose that the last vertex of c is a vertex $u \neq v$. As $v \in S_G$ by assumption, there exists a vertex $w \neq v$ such that $N[v] \subseteq N[w]$. If v does not give birth to any vertex in c then we can move v to the end of c, i.e., right after vertex u. Let c' be the resulting candidate elimination ordering of G; then $c' \setminus v$ is a candidate elimination ordering of $G - v$, as the parent-child relations of $G - v$ are the same in both $c' \setminus v$ and c.

Finally suppose that v gives birth to at least one vertex, and let Z be the set of vertices which are born by v or by some descendant of v. If w appears before v in c, then for any vertex in Z we assign its parent to be w (instead of v). Note that this is always possible as $N[v] \subseteq N[w]$. Now suppose that w appears after v in c, and let $Z_0 = \{z \in Z : v <_c z <_c w\}$ be the vertices of Z which lie between v and w in c. Then we move all vertices of Z_0 immediately after w (without changing their relative order). Finally, similarly to the above, for any vertex in Z we assign its parent to be w (instead of v). In either case (i.e., when w is before or after v in c), after making these changes we obtain a candidate elimination ordering c'' of G, in which v does not give birth to any other vertex. Thus we can obtain from c'' a new candidate elimination ordering c'''

of G where v is moved to the end of the ordering. Then $c''' \setminus v$ is a candidate elimination ordering of $G - v$, as the parent-child relations of $G - v$ are the same in both $c''' \setminus v$ and c''. □

Theorem 3.17. *The* `candidate elimination ordering` *algorithm is a polynomial-time algorithm that, for any graph G, decides whether G has a growth schedule of $n-1$ slots and $\ell = 0$ excess edges, and it outputs such a schedule if one exists.*

Proof. First note that we can find the candidate vertices in polynomial time, and thus, the algorithm terminates in polynomial time. This is because the algorithm removes one candidate vertex u in each loop, which based on Lemma 3.16. This means that the algorithm terminates with a growth schedule for G if one exists. □

The notion of candidate elimination orderings turns out to coincide with the notion of cop-win orderings, discovered in the past in graph theory for a class of graphs, called cop-win graphs [80–82]. In particular, it is not hard to show that *a graph has a candidate elimination ordering if and only if it is a cop-win graph.* This implies that our *`candidate elimination ordering`* algorithm is probably equivalent to some folklore algorithms in the literature of cop-win graphs.

Lemma 3.18 ($\star$)**.** *There is a modified version of the candidate elimination ordering algorithm that computes in polynomial time a growth schedule for any graph G of $n-1$ slots and ℓ excess edges, where ℓ is a constant, if and only if such a schedule exists.*

Proof. The candidate elimination ordering algorithm can be slightly modified to check whether a graph $G = (V, E)$ has a growth schedule of $n-1$ slots and ℓ excess edges. The modification is quite simple. For $\ell = 1$, we create multiple graphs G'_x for $x = 1, 2, \ldots, e^2 - |E|$ where each graph G'_x is the same as G with the addition of one arbitrary edge $e \notin E$, and we do this for all possible edge additions. Since the complement of G has $e \leq n^2$ edges, we will create up to n^2 graphs G'_x. In essence, $G'_x = (V'_x, E'_x)$, where $V'_x = V$ and $E'_x = E \cup uv$ such that $uv \notin E$ and $(E'_j \neq E'_i)$, for all $i \neq j$. We then run the candidate elimination ordering algorithm on all G'_x. If the algorithm returns "no" for all of them, then there exists no growth schedule for G of $n-1$ slots and 1 excess edge. Otherwise, the algorithm outputs a schedule of $n-1$ slots and 1 excess edge for graph G. This process can be modified for any ℓ, but then the number of graphs G'_x tested are $n^{2\ell}$. Therefore ℓ has to be a constant in order to check all graphs G'_x in polynomial time. □

Our next goal is to decide whether a graph $G = (V, E)$ on n vertices has a growth schedule σ of $\log n$ slots and $\ell = 0$ excess edges. For easiness of notation, we assume that $n = 2^\delta$ for some integer δ. This assumption can be easily removed. The *`fast growth`* algorithm computes the slots of the growth schedule in reverse order.

`Fast growth` algorithm: The algorithm finds set S_G of candidate vertices in G. It then tries to find a subset $L \subseteq S_G$ of candidates that satisfies all of the following:

1. $|L| = n/2$.
2. L is an independent set.
3. There is a perfect matching between the candidate vertices in L and their candidate parents in G.

Any set L that satisfies the above constraints is called *valid*. The algorithm finds such a set by creating a 2-SAT formula ϕ whose solution is a valid set L. If the algorithm finds such a set L, it adds the vertices in L to the last slot of the schedule. It then removes the vertices in L from graph G along with their incident edges. The above process is then repeated to find the next slots. If at any point, graph G has a single vertex, the algorithm terminates and outputs the schedule. If at any point, the algorithm cannot find a valid set L, it outputs "no".

Lemma 3.19. *Consider any graph $G = (V, E)$. If G has a growth schedule of $\log n$ slots and $\ell = 0$ excess edges then there exists a perfect matching M that contains a valid candidate vertex set L, where L has exactly one vertex for each edge of the perfect matching M.*

Proof. Let us assume that graph G has a growth schedule. Then in the last slot, there are $n/2$ vertices, called parents, for which $n/2$ other vertices, called children, are generated. Therefore, such a perfect matching M always exists where set L contains the children. □

Lemma 3.20. *The 2-SAT formula ϕ, generated by the* `fast growth algorithm`*, has a solution if and only if there is an independent set $|V_2| = n/2$, where V_2 is a valid set of candidate vertices in graph $G = (V, E)$.*

Proof. Let us assume that graph G has a growth schedule. Based on Lemma 3.19, there are $n/2$ parents and $n/2$ children in G. Therefore, there has to be a set V_2, where $|V_2| = n/2$ and V_2 is an independent set such that there is another set V_1, where $|V_1| = n/2$ and $V_1 \cap V_2 = \emptyset$. Any perfect matching $M \in G$ includes edges $u_i v_i \in M$, where $u_i \in V_1$ and $v_i \in V_2$ because V_2 is an independent set.

The solution to the 2-SAT formula ϕ we are going to create is a valid set V_2 as stated above. Consider an arbitrary edge $u_i v_i$ from the perfect matching M. The algorithm creates a variable x_i for each $u_i v_i$. The truthful assignment of x_i means that we pick v_i for V_2 and the negative assignment means that we pick u_i for V_2. Since $|V_2| = n/2$, then for every edge $u_i v_i \in M$, at least one of u_i, v_i is a candidate vertex, because otherwise some other edge $u_j v_j \in M$ would need to have 2 candidates vertices at its endpoints

Algorithm 9 Fast growth algorithm

Input: A graph $G = (V, E)$ on $n = 2^\delta$ vertices.
Output: A growth schedule of $k = \log n$ slots and $\ell = 0$ excess edges for G.

for $k = \log n$ downto 1 **do**
  $\mathcal{S}_k = \emptyset$; $\phi = \emptyset$
  Find a perfect matching $M = \{u_i v_i : 1 \leq i \leq n/2\}$ of G.
  if No perfect matching exists **then**
    return "NO"
  for every edge $u_i v_i \in M$ **do**
    Create variable x_i
  for every edge $u_i v_i \in M$ **do**
    if $(N[u_i] \subseteq N[w]$, for some vertex $w \in V \setminus \{u_i\}) \wedge (N[v_i] \not\subseteq N[x]$, for any vertex $x \in V \setminus \{v_i\})$ **then** $\{u_i$ is a candidate vertex and v_i is not.$\}$
      $\phi \leftarrow \phi \wedge (\neg x_i)$
    else if $(N[u_i] \not\subseteq N[w]$ for any vertex $w \in V \setminus \{u_i\}) \wedge (N[v_i] \subseteq N[x]$, for some vertex $x \in V \setminus \{v_i\})$ **then** $\{u_i$ is a not candidate and v_i is a candidate$\}$
      $\phi \leftarrow \phi \wedge (x_i)$
    else if $(N[u_i] \not\subseteq N[w]$, for some vertex $w \in V \setminus \{u_i\}) \wedge (N[v_i] \not\subseteq N[x]$, for some vertex $x \in V \setminus \{v_i\})$ **then** $\{u_i$ is not a candidate and v_i is not a candidate$\}$
      return "NO"
  for every edge $u_i u_j \in E \setminus M$ **do**
    $\phi \leftarrow \phi \wedge (x_i \vee x_j)$
  for every edge $v_i v_j \in E \setminus M$ **do**
    $\phi \leftarrow \phi \wedge (\neg x_i \vee \neg x_j)$
  for every edge $u_i v_j \in E \setminus M$ **do**
    $\phi \leftarrow \phi \wedge (x_i \vee \neg x_j)$
  if ϕ is satisfiable **then**
    Let τ be a satisfying truth assignment for ϕ
    for $i = 1, 2, \ldots, n/2$ **do**
      if $x_i = true$ in τ **then**
        $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup (u_i, v_i, \{v_i w : w \in N(v_i)\})$
        $V \leftarrow V \setminus \{v_i\}$
        $E \leftarrow E \setminus \{v_i w : w \in N(v_i)\}$
      else $\{x_i = false$ in $\tau\}$
        $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup (v_i, u_i, \{u_i w : w \in N(u_i)\})$
        $V \leftarrow V \setminus \{u_i\}$
        $E \leftarrow E \setminus \{u_i w : w \in N(u_i)\}$
  else $\{\phi$ is not satisfiable$\}$
    return "NO"
return $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k, \emptyset)$

and include them both in V_2, which is impossible. Thus, graph G would have no growth schedule.

If v_i is a candidate vertex and u_i is not, then $v_i \in V_2$, and we add clause (x_i) to ϕ. If u_i is a candidate vertex and v_i is not, then $u_i \in V_2$, in which case we add clause

$(\neg x_i)$ ϕ. If both u_i and v_i are candidate vertices, either one could be in V_2 as long as V_2 is an independent set.

We now want to make sure that every vertex in V_2 is independent. Therefore, for every edge $u_iu_j \in E$, we add clause $(x_i \vee x_j)$ to ϕ. This means that in order to satisfy that clause, u_i and u_j cannot be both picked for V_2. Similarly, for every edge $v_iv_j \in E$, we add clause $(\neg x_i) \vee (\neg x_j)$ to ϕ and for every edge $u_iv_j \in E$, we add clause $(x_i) \vee (\neg x_j)$ to ϕ.

The solution to formula ϕ is a valid set V_2 and we can find it in polynomial time. If the formula has no solution, then no valid independent set V_2 exists for graph G. □

Lemma 3.21. *Consider any graph $G = (V, E)$. If G has a growth schedule of $\log n$ slots and $\ell = 0$ excess edges, then any arbitrary perfect matching contains a valid candidate set $|L| = n/2$, where L has exactly one vertex for each edge of the perfect matching.*

Proof. By Lemma 3.20, any perfect matching M contains edges uv, such that there exists a valid candidate set V_2 that contains one vertex exactly for each edge $uv \in M$. Thus, if graph G has a growth schedule, the solution to the 2-SAT formula corresponds to a valid candidate set V_2. □

Theorem 3.22. *For any graph G on 2^δ vertices, the* `fast growth` *algorithm computes in polynomial time a growth schedule σ for G of $\log n$ slots and $\ell = 0$ excess edges, if one exists.*

Proof. Suppose that $G = (V, E)$ has a growth schedule σ of $\log n$ slots and $\ell = 0$ excess edges. By Lemmas 3.20 and 3.21 we know that our ***`fast growth`*** algorithm finds a set L for the last slot of a schedule σ'' but this might be a different set from the last slot contained in σ. Therefore, for our proof to be complete, we need to show that if G has a growth schedule σ of $\log n$ slots and $\ell = 0$ excess edges, for any L it holds that $(G - L)$ has a growth schedule σ' of $\log n - 1$ slots and $\ell = 0$ excess edges.

Assume that σ has in the last slot $\mathcal{S}_k$ a set of vertices V_1 generating another set of vertices V_2, such that $|V_1| = |V_2| = n/2$, $V_1 \cap V_2 = \emptyset$ and V_2 is an independent set. Suppose that our algorithm finds V_2' such that $V_2' \neq V_2$.

Assume that $V_2' \cap V_2 = V_s$ and $|V_s| = n/2 - 1$. This means that $V_2' = V_s \cup u'$ and $V_2 = V_s \cup u$ and u' has no edge with any vertex in V_s. Since $u' \notin V_2$ and u' has no edge with any vertex in V_s, then $u' \in V_1$. However, u' cannot be the candidate parent of anyone in V_2 apart from u. Similarly, u is the only candidate parent of u'. Therefore $N[u] \subseteq N[u'] \subseteq N[u] \implies N[u] = N[u']$. This means that we can swap the two vertices in any growth schedule and still maintain a correct growth schedule for G. Therefore, for $L = V_2'$, the graph $(G - L)$ has a growth schedule σ' of $\log n - 1$ slots and $\ell = 0$ excess edges.

Assume now that $V_2' \cap V_2 = V_s$, where $|V_s| = x \geq 0$. Then, $V_2' = V_s \cup u_1' \cup u_2', \cup \ldots \cup u_y'$ and $V_2 = V_s \cup u_1 \cup u_2, \cup \ldots \cup u_y$, where $y = n/2 - x$. As argued above, vertices $u_1', u_2', \ldots, u_y'$ can be candidate parents only to vertices $u_1, u_2, \ldots, u_y$, and vice versa. Thus, there is a pairing u_j, u_j' such that $N[u_j] \subseteq N[u_j'] \subseteq N[u_j] \implies N[u_j'] = N[u_j]$, for every $j = 1, 2, \ldots, y$. Thus, these vertices can be swapped in the growth schedule and still maintain a correct growth schedule for G. Therefore for any arbitrary $L = V_2'$, the graph $(G - L)$ has a growth schedule σ' of $\log n - 1$ slots and $\ell = 0$ excess edges. □

We will now show that the problem of computing the minimum number of slots required for a graph G to be grown is NP-complete, and that it cannot be approximated within a $n^{\frac{1}{3}-\varepsilon}$ factor for any $\varepsilon > 0$, unless P = NP.

Definition 3.23. Given any graph G and a natural number κ, find a growth schedule of κ slots and $\ell = 0$ excess edges. We call this problem *zero-excess growth schedule.*

Theorem 3.24. *The decision version of the zero-excess graph growth problem is NP-complete.*

Proof. First, observe that the decision version of the problem belongs to the class NP. Indeed, the required polynomial certificate is a given growth schedule σ, together with an isomorphism between the graph constructed by σ and the target graph G.

To show NP-hardness, we provide a reduction from the coloring problem. Given an arbitrary graph $G = (V, E)$ on n vertices, we construct graph $G' = (V', E')$ as follows: Let $G_1 = (V_1, E_1)$ be an isomorphic copy of G, and let G_2 be a clique of n vertices. G' consists of the union of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where we also add all possible edges between them. Note that every vertex of G_2 is a universal vertex in G' (i.e., a vertex which is connected with every other vertex in the graph). Let $\chi(G)$ be the chromatic number of graph G, and let $\kappa(G')$ be the minimum number of slots required for a growth schedule for G'. We will show that $\kappa(G') = \chi(G) + n$.

Let σ be an optimal growth schedule for G', which uses $\kappa(G')$ slots. As every vertex $v \in V_2$ is a universal vertex in G', v cannot coexist with any other vertex of G' in the same slot of σ. Furthermore, the vertices of V_1 require at least $\chi(G)$ different slots in σ, since $\chi(G)$ is the smallest possible partition of V_1 into independent sets. Thus $\kappa(G') \geq \chi(G) + n$.

We now provide the following growth schedule σ^* for G', which consists of exactly $\chi(G)+n$ slots. Each of the first n slots of σ^* contains exactly one vertex of V_2; note that each of these vertices (apart from the first one) can be generated and connected with an earlier vertex of V_2. In each of the following $\chi(G)$ slots, we add one of the $\chi(G) = \chi(G_1)$ color classes of an optimal coloring of G_1. Consider an arbitrary color class of G_1 and suppose that it contains p vertices; these p vertices can be born by exactly p of the universal vertices of V_2 (which have previously appeared in σ^*). This completes the growth schedule σ^*. Since σ^* has $\chi(G) + n$ slots, it follows that $\kappa(G') \leq \chi(G) + n$. □

Theorem 3.25. *Let $\varepsilon > 0$. If there exists a polynomial-time algorithm, which, for every graph G, computes a $n^{\frac{1}{3}-\varepsilon}$-approximate growth schedule (i.e., a growth schedule with at most $n^{\frac{1}{3}-\varepsilon}\kappa(G)$ slots), then $P = NP$.*

Proof. The reduction is from the minimum coloring problem. Given an arbitrary graph $G = (V, E)$ with n vertices, we construct in polynomial time a graph $G' = (V', E')$ with $N = 4n^3$ vertices, as follows: We create $2n^2$ isomorphic copies of G, which are denoted by $G_1^A, G_2^A, \ldots, G_{n^2}^A$ and $G_1^B, G_2^B, \ldots, G_{n^2}^B$, and we also add n^2 clique graphs, each of size $2n$, denoted by $C_1, C_2, \ldots, C_{n^2}$. We define $V' = V(G_1^A) \cup \ldots \cup V(G_{n^2}^A) \cup V(G_1^B) \cup \ldots \cup V(G_{n^2}^B) \cup V(C_1) \cup \ldots \cup V(C_{n^2})$. Initially we add to the set E' the edges of all graphs $G_1^A, \ldots, G_{n^2}^A$, $G_1^B, \ldots, G_{n^2}^B$, and $C_1, \ldots, C_{n^2}$. For every $i = 1, 2, \ldots, n^2 - 1$ we add to E' all edges between $V(G_i^A) \cup V(G_i^B)$ and $V(G_{i+1}^A) \cup V(G_{i+1}^B)$. For every $i = 1, \ldots, n^2$, we add to E' all edges between $V(C_i)$ and $V(G_i^A) \cup V(G_i^B)$. Furthermore, for every $i = 2, \ldots, n^2$, we add to E' all edges between $V(C_i)$ and $V(G_{i-1}^A) \cup V(G_{i-1}^B)$. For every $i = 1, \ldots, n^2 - 1$, we add to E' all edges between $V(C_i)$ and $V(C_{i+1})$. For every $i = 1, 2, \ldots, n^2$ and for every $u \in V(G_i^B)$, we add to E' the edge uu', where $u' \in V(G_i^A)$ is the image of u in the isomorphism mapping between G_i^A and G_i^B. To complete the construction, we pick an arbitrary vertex a_i from each C_i. We add edges among the vertices $a_1, \ldots, a_{n^2}$ such that the resulting induced graph $G'[a_1, \ldots, a_{n^2}]$ is a graph on n^2 vertices which can be grown by a ***path*** schedule within $\lceil \log n^2 \rceil$ slots and with zero excess edges (see Lemma 3.2[3]). This completes the construction of G'. Clearly, G' can be constructed in time polynomial in n.

Now we will prove that there exists a growth schedule σ' of G' of length at most $n^2\chi(G) + 4n - 2 + \lceil 2 \log n \rceil$. The schedule will be described inversely, that is, we will describe the vertices generated in each slot starting from the last slot of σ' and finishing with the first slot. First note that every $u \in V(G_{n^2}^A) \cup V(G_{n^2}^B)$ is a candidate vertex in G' Indeed, for every $w \in V(C_{n^2})$, we have that $N[u] \subseteq V(G_{n^2}^A) \cup V(G_{n^2}^B) \cup V(G_{n^2-1}^A) \cup V(G_{n^2-1}^A) \cup V(C_{n^2}) \subseteq N[w]$. To provide the desired growth schedule σ', we assume that a minimum coloring of the input graph G (with $\chi(G)$ colors) is known. In the last $\chi(G)$ slots, σ' generates all vertices in $V(G_{n^2}^A) \cup V(G_{n^2}^B)$, as follows. At each of these slots, one of the $\chi(G)$ color classes of the minimum coloring c_{OPT} of $G_{n^2}^A$ is generated on sufficiently many vertices among the first n vertices of the clique C_{n^2}. Simultaneously, a different color class of the minimum coloring c_{OPT} of $G_{n^2}^B$ is generated on sufficiently many vertices among the last n vertices of the clique C_{n^2}.

Similarly, for every $i = 1, \ldots, n^2 - 1$, once the vertices of $V(G_{i+1}^A) \cup \ldots \cup V(G_{n^2}^A) \cup V(G_{i+1}^B) \cup \ldots \cup V(G_{n^2}^B)$ have been added to the last $(n^2 - i)\chi(G)$ slots of σ', the

[3]From Lemma 3.2 it follows that the path on n^2 vertices can be constructed in $\lceil \log n^2 \rceil$ slots using $O(n^2)$ excess edges. If we put all these $O(n^2)$ excess edges back to the path of n^2 vertices, we obtain a new graph on n^2 vertices with $O(n^2)$ edges. This graph is the induced subgraph $G'[a_1, \ldots, a_{n^2}]$ of G' on the vertices $a_1, \ldots, a_{n^2}$.

vertices of $V(G_i^A) \cup V(G_i^B)$ are generated in σ' in $\chi(G)$ more slots. This is possible because every vertex $u \in V(G_i^A) \cup V(G_i^B)$ is a candidate vertex after the vertices of $V(G_{i+1}^A) \cup \ldots \cup V(G_{n^2}^A) \cup V(G_{i+1}^B) \cup \ldots \cup V(G_{n^2}^B)$ have been added to slots. Indeed, for every $w \in V(C_i)$, we have that $N[u] \subseteq V(G_i^A) \cup V(G_i^B) \cup V(G_{i-1}^A) \cup V(G_{i-1}^A) \cup V(C_i) \subseteq N[w]$. That is, in total, all vertices of $V(G_1^A) \cup \ldots \cup V(G_{n^2}^A) \cup V(G_1^B) \cup \ldots \cup V(G_{n^2}^B)$ are generated in the last $n^2\chi(G)$ slots.

The remaining vertices of $V(C_1) \cup \ldots \cup V(C_{n^2})$ are generated in σ' in $4n-2+\lceil \log n^2 \rceil$ additional slots. First, for every odd index i and for $2n-1$ consecutive slots, for vertex a_i of $V(C_i)$ exactly one other vertex of $V(C_i)$ is generated. This is possible because for every vertex $u \in V(C_i) \setminus a_i$, $N[u] \subseteq V(C_i) \cup V(C_{i-1}) \cup V(C_{i+1}) \subseteq N[a_i]$. Then, for every even index i and for $2n-1$ further consecutive slots, for vertex a_i of $V(C_i)$ exactly one other vertex of $V(C_i)$ is generated. That is, after $4n-2$ slots only the induced subgraph of G' on the vertices $a_1, \ldots, a_{n^2}$ remains. The final $\lceil \log n^2 \rceil$ slots of σ' are the ones obtained by Lemma 3.2. To sum up, G' is grown by the growth schedule σ' in $k = n^2\chi(G) + 4n - 2 + \lceil \log n^2 \rceil$ slots, and thus $\kappa(G') \leq n^2\chi(G) + 4n - 2 + \lceil 2 \log n \rceil$ (1).

Suppose that there exists a polynomial-time algorithm A which computes an $N^{\frac{1}{3}-\varepsilon}$-approximate growth schedule σ'' for graph G' (which has N vertices), i.e., a growth schedule of $k \leq N^{\frac{1}{3}-\varepsilon}\kappa(G')$ slots. Note that, for every slot of σ'', all different vertices of $V(G_i^A)$ (resp. $V(G_i^B)$) which are generated in this slot are independent. For every $i = 1, \ldots, n^2$, denote by χ_i^A (resp. χ_i^B) the number of different slots of σ'' in which at least one vertex of $V(G_i^A)$ (resp. $V(G_i^B)$) appears. Let $\chi^* = \min\{\chi_i^A, \chi_i^B : 1 \leq i \leq n^2\}$. Then, there exists a coloring of G with at most χ^* colors (i.e., a partition of G into at most χ^* independent sets).

Now we show that $k \geq \frac{1}{2}n^2\chi^*$. Let $i \in \{2, \ldots, n^2 - 1\}$ and let $u \in V(G_i^A) \cup V(G_i^B)$. Assume that u is generated at slot t in σ''. Then, either all vertices of $V(G_{i-1}^A) \cup V(G_{i-1}^B)$ or all vertices of $V(G_{i+1}^A) \cup V(G_{i+1}^B)$ are generated at a later slot $t' \geq t+1$ in σ''. Indeed, it can be easily checked that, if otherwise both a vertex $x \in V(G_{i-1}^A) \cup V(G_{i-1}^B)$ and a vertex $y \in V(G_{i+1}^A) \cup V(G_{i+1}^B)$ are generated at a slot $t'' \leq t$ in σ'', then u cannot be a candidate vertex at slot t, which is a contradiction to our assumption. That is, in order for a vertex $u \in V(G_i^A) \cup V(G_i^B)$ to be generated at some slot t of σ'', we must have that i is either the currently smallest or largest index for which some vertices of $V(G_i^A) \cup V(G_i^B)$ have been generated until slot t. On the other hand, by definition of χ^*, the growth schedule σ'' needs at least χ^* different slots to generate all vertices of the set $V(G_i^A) \cup V(G_i^B)$, for $1 \leq i \leq n^2$. Therefore, since at every slot, σ'' can potentially generate vertices of at most two indices i (the smallest and the largest respectively), it needs to use at least $\frac{1}{2}n^2\chi^*$ slots to grow the whole graph G'. Therefore $k \geq \frac{1}{2}n^2\chi^*$ (2).

Recall that $N = 4n^3$. It follows by Eq. (1) and Eq. (2) that

$$\begin{aligned}\frac{1}{2}n^2\chi^* &\leq k \leq N^{\frac{1}{3}-\varepsilon}\kappa(G') \\ &\leq N^{\frac{1}{3}-\varepsilon}(n^2\chi(G) + 4n - 2 + \lceil 2\log n \rceil) \\ &\leq 4n^{1-3\varepsilon}(n^2\chi(G) + 6n)\end{aligned}$$

and thus $\chi^* \leq 8n^{1-3\varepsilon}\chi(G) + 48n^{-3\varepsilon}$. Note that, for sufficiently large n, we have that $8n^{1-3\varepsilon}\chi(G) + 48n^{-3\varepsilon} \leq n^{1-\varepsilon}\chi(G)$. That is, given the $N^{\frac{1}{3}-\varepsilon}$-approximate growth schedule produced by the polynomial-time algorithm A, we can compute in polynomial time a coloring of G with χ^* colors such that $\chi^* \leq n^{1-\varepsilon}\chi(G)$. This is a contradiction since for every $\varepsilon > 0$, there is no polynomial-time $n^{1-\varepsilon}$-approximation for minimum coloring, unless P = NP [88]. □ □

3.4 Growth Schedules of (Poly)logarithmic Slots

In this section, we study graphs that have growth schedules of (poly)logarithmic slots, for $d = 2$. As we have proven in the previous section, an integral factor in computing a growth schedule for any graph G, is computing a k-coloring for G. Since we consider polynomial-time algorithms, we have to restrict ourselves to graphs where the k-coloring problem can be solved in polynomial time and, additionally, we want small values of k since we want to produce fast growth schedules. Therefore, we investigate tree, planar and k-degenerate graph families since there are polynomial-time algorithms that solve the k-coloring problem for graphs drawn from these families. We continue with lower bounds on the number of excess edges if we fix the number of slots to $\log n$, for path, star and specific bipartite graph families.

3.4.1 Trees

We now provide an algorithm that computes growth schedules for tree graphs. Let G be the target tree graph. The algorithm applies a decomposition strategy on G, where vertices and edges are removed in phases, until a single vertex is left. We can then grow the target graph G by reversing its decomposition phases, using the ***path*** and ***star*** schedules as subroutines.

***Tree* algorithm**: Starting from a tree graph G, the algorithm keeps alternating between two phases, a *path-cut* and a *leaf-cut* phase. Let G_{2i}, G_{2i+1}, for $i \geq 0$, be the graphs obtained after the execution of the first i pairs of phases and an additional path-cut phase, respectively.

Path-cut phase: For each path subgraph $P = (u_1, u_2, \ldots, u_\nu)$, for $2 < \nu \leq n$, of the current graph G_{2i}, where $u_2, u_3, ..., u_{\nu-1}$ have degree 2 and u_1, u_ν have degree $\neq 2$

in G_{2i}, edge u_1u_ν between the endpoints of P is activated and vertices $u_2, u_3, ...u_{\nu-1}$ are removed along with their incident edges. If a single vertex is left, the algorithm terminates; otherwise, it proceeds to the leaf-cut phase.

Leaf-cut phase: Every leaf vertex of the current graph G_{2i+1} is removed along with its incident edge. If a single vertex is left, the algorithm terminates; otherwise, it proceeds to the path-cut phase.

Finally, the algorithm reverses the phases (by decreasing i) to output a growth schedule for the tree G as follows. For each path-cut phase $2i$, all path subgraphs that were decomposed in phase i are regrown by using the ***path*** schedule as a subprocess. These can be executed in parallel in $O(\log n)$ slots. The same holds true for leaf-cut phases $2i + 1$, where each can be reversed to regrow the removed leaves by using ***star*** schedules in parallel in $O(\log n)$ slots. In the last slot, the schedule deletes every excess edge.

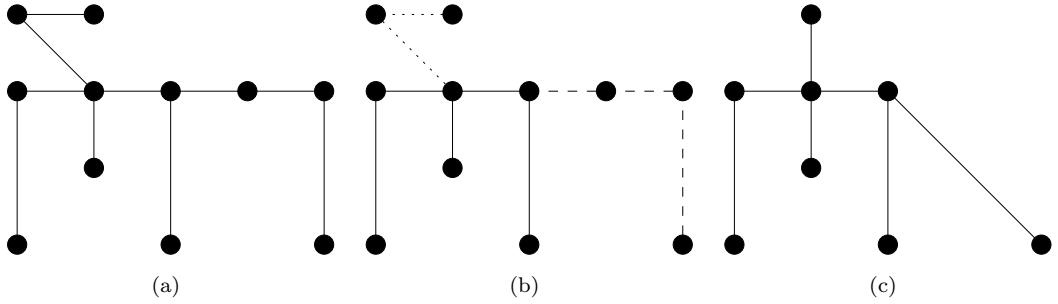

FIGURE 3.4: An example of a path-cut phase. (a) Graph G_{2i-1} at the beginning of the i-th path-cut phase. (b) The dotted and the dashed edges with their incident vertices form two different path subgraphs. Every vertex, apart from the endpoints of each path, is removed and the endpoints become connected. (c) The resulting graph G_{2i+1} at the end of the i-th path-cut phase.

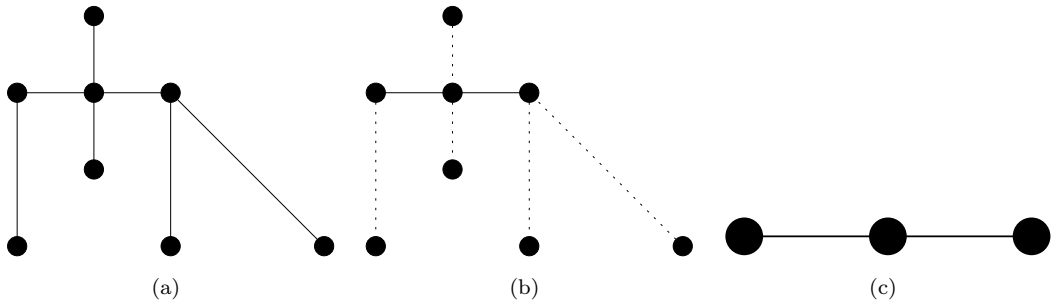

FIGURE 3.5: An example of a leaf-cut phase. (a) Graph G_{2i+1} at the beginning of the i-th leaf-cut phase. (b) The leaf vertices along with their incident edges are removed. (c) The resulting graph G_{2i+2} at the end of the i-th leaf-cut phase.

Lemma 3.26. *Given any tree graph G, the algorithm deconstructs G into a single vertex using $O(\log n)$ phases.*

Proof. Consider the graph G_{2i} after the execution of the i-th path-cut phase. The path-cut phase removes every vertex that has exactly 2 neighbors in the current graph, and in the next leaf cut phase, the graph consists of leaf vertices $u \in S_u$ with $deg = 1$ and internal vertices $v \in S_v$ with $deg > 2$. Therefore, $|S_u| > |S_v|$ and since $|S_u| + |S_v| = |V_i|$, we can conclude that $|S_u| > |V_i|/2$ and any leaf-cut phase cuts the size of the current graph in half since it removes every vertex $u \in S_u$. This means that after at most $\log n$ path-cut phases and $\log n$ leaf cut phases the graph will have a single vertex. □

Lemma 3.27. *Every phase can be reversed using a growth schedule of $O(\log n)$ slots.*

Proof. First, let us consider the path-cut phase. At the beginning of this phase, every starting subgraph G' is a path subgraph with vertices $u_1, u_2, ..., u_x$, where u_1, u_x are the endpoints of the path. At the end of the phase, every subgraph has two connected vertices u_1, u_x. The reversed process works as follows: for each path $u_1, u_2, \dots u_x$ that we want to generate, we use vertex u_1 as the initiator and we execute the ***path*** algorithm from Section 3.2.2 in order to generate vertices $u_2, u_3, ..., u_{x-1}$. We add the following modification to ***path***: every time a vertex is generated, an edge between it and vertex u_x is activated. After this process completes, edges not belonging to the original path subgraph G' are deleted. This growth schedule requires $\log x \leq \log n$ slots. We can combine the growth schedules of each path into a single schedule of $\log x$ slots since every schedule has distinct initiators and they can run in parallel.

Now let us consider the leaf-cut phase. In this phase, every vertex removed is a leaf vertex u with one neighbor v. Note that v might have multiple neighboring leaves. The reverse process works as follows: For each vertex v, we use a separate star growth schedule from Section 3.2.2 with v as the initiator, in order to generate every vertex u that was a neighbor to v. Each of this growth schedule requires at most $\log x \leq \log n$ slots, where x is the number of leaves in the current graph. We can combine the growth schedules of each star into a single schedule of $\log k$ slots since every schedule has distinct initiators and they can run in parallel. □

Theorem 3.28. *For any tree graph G on n vertices, the* `tree` *algorithm computes in polynomial time a growth schedule σ for G of $O(\log^2 n)$ slots and $O(n)$ excess edges.*

Proof. The growth schedules can be straightly combined into a single one by appending the end of each growth schedule with the beginning of the next one, since every sub-schedule σ_i uses only a single vertex as an initiator u, which is always available (i.e., u was generated by some previous σ_j). Since we have $O(\log n)$ schedules and every schedule has $O(\log n)$ slots, the combined growth schedule has $O(\log^2 n)$ slots. Note

that every schedule used to reverse a phase uses $O(n)$ excess edges, where n is the number of vertices generated in that schedule. Since the complete schedule generates $n-1$ vertices, the excess edges activated throughout the complete schedule are $O(n)$. □

3.4.2 Planar Graphs

In this section, we provide an algorithm that computes a growth schedule for any target planar graph $G = (V, E)$. The algorithm first computes a 5-coloring of G and partitions the vertices into color-sets V_i, $1 \leq i \leq 5$. The color-sets are used to compute the growth schedule for G. The schedule contains five sub-schedules, each sub-schedule i generating all vertices in color-set V_i. In every sub-schedule i, we use a modified version of the *star* schedule to generate set V_i.

Pre-processing: By using the algorithm of [89], the pre-processing step computes a 5-coloring of the target planar graph G. This creates color-sets $V_i \subseteq V$, where $1 \leq i \leq 5$, every color-set V_i containing all vertices of color i. W.l.o.g., we can assume that $|V_1| \geq |V_2| \geq |V_3| \geq |V_4| \geq |V_5|$. Note that every color-set V_i is an independent set of G.

Planar **algorithm**: The algorithm picks an arbitrary vertex from V_1 and makes it the initiator u_0 of all sub-schedules. Let $V_i = \{u_1, u_2, \ldots, u_{|V_i|}\}$. For every sub-schedule i, $1 \leq i \leq 5$, it uses the *star* schedule with u_0 as the initiator, to grow the vertices in V_i in an arbitrary sequence, with some additional edge activations. In particular, upon generating vertex $u_x \in V_i$, for all $1 \leq x \leq |V_i|$:

1. Edge vu_x is activated if $v \in \bigcup_{j<i} V_j$ and $u_y v \in E$, for some $u_y \in V_i \cap P_{u_x}$, both hold (recall that P_{u_x} contains the descendants of u_x).
2. Edge wu_x is activated if $w \in \bigcup_{j<i} V_j$ and $wu_x \in E$ both hold.

Once all vertices of V_i have been generated, the schedule moves on to generate V_{i+1}. Once all vertices have been generated, the schedule deletes every edge $uv \notin E$. Note that every edge activated in the growth schedule is an excess edge with the exception of edges satisfying (2). For an edge wu_x from (2) to satisfy the edge-activation distance constraint it must hold that every vertex in the birth path of u_x has an edge with w. This holds true for the edges added in (2), due to the edges added in (1).

The edges of the *star* schedule are used to quickly generate the vertices, while the edges of (1) are used to enable the activation of the edges of (2). By proving that the *star* schedule activate $O(n)$ edges, (1) activates $O(n \log n)$ edges, and by observing that the schedule contains *star* sub-schedules that have $5 \times O(\log n)$ slots in total, the next theorem follows.

Lemma 3.29. *Given a target planar graph $G = (V, E)$, the* `planar` *algorithm returns a growth schedule for G.*

Proof. Based on the description of the schedule, it is easy to see that we generate exactly $|V|$ vertices, since we break V into our five sets V_i and we generate each set in a different phase i. This is always possible no matter the graph G, since every set V_i is an independent set.

We will now prove that we also generate activate the edges of G. Note that this holds trivially since (2) activates exactly those edges. What remains is to argue that the edges of (2) do not violate the edge activation distance $d = 2$ constraint. This constraint is satisfied by the edges activated by (1) since for every edge $wu_x \in G$, the schedule makes sure to activate every edge uu_y, where vertices u_y are the vertices in the birth path of u_x. □

Lemma 3.30. *The* `planar` *algorithm has* $O(\log n)$ *slots and* $O(n \log n)$ *excess edges.*

Proof. Let n_i be the size of the independent set V_i. Then, the sub-schedule that constructs V_i requires the same number of slots as ***`path`***, which is $\lceil \log n_i \rceil$ slots. Combining the five sub-schedules requires $\sum_{i=1}^{5} \log n_i = \log \prod_{i=1}^{5} n_i < 5 \log n = O(\log n)$ slots.

Let us consider the excess edges activated in every sub-schedule. The number of excess edges activated are the excess edges of the star schedule and the excess edges for the progeny of each vertex. The excess edges of the star schedule are $O(n)$. We also know that the progeny of each vertex u includes at most $|P_u| = O(\log n)$ vertices since the length of the growth schedule is $O(\log n)$. Since we have a planar graph we know that there are at most $3n$ edges in graph G. For every edge (u, v) in the target graph, we would need to add at most $O(\log n)$ additional excess edges. Therefore, no matter the structure of the $3n$ edges, the schedule would activate $3nO(\log n) = O(n \log n)$ excess edges. □

The next theorem now follows from Lemmas 3.29 and 3.30.

Theorem 3.31. *For any planar graph* G *on* n *vertices, the* `planar` *algorithm computes in polynomial time a growth schedule for* G *of* $O(\log n)$ *slots and* $O(n \log n)$ *excess edges.*

Definition 3.32. A k-degenerate graph G is an undirected graph in which every subgraph has a vertex of degree at most k.

Corollary 3.33. *The* `planar` *algorithm can be extended to compute, for any graph* G *on* n *vertices and in polynomial time, a growth schedule of* $O((k_1 + 1) \log n)$ *slots,* $O(k_2 n \log n)$ *and excess edges, where (i)* $k_1 = k_2$ *is the degeneracy of graph* G*, or (ii)* $k_1 = \Delta$ *is the maximum degree of graph* G *and* $k_2 = |E|/n$*.*

Proof. For case (i), if graph G is k_1*-degenerate*, then an ordering with coloring number $k_1 + 1$ can be obtained by repeatedly finding a vertex v with at most x neighbors, removing v from the graph, ordering the remaining vertices, and adding v to the end

of the ordering. By Lemma 3.30, the algorithm using a $k_1 + 1$ coloring would produce a growth schedule of $O((k_1 + 1) \log n)$ slots. Since graph G is $k_2 - degenerate$, G has at most $k_2 \times n$ edges and by the proof of Lemma 3.30, the algorithm would require $O(k_2 n \log n)$ excess edges. For case (ii), we compute a $\Delta + 1$ coloring using a greedy algorithm and then use the planar graph algorithm with the computed coloring as an input. By the proof of Lemma 3.30, the algorithm would produce a growth schedule of $O((\Delta + 1) \log n)$ slots. □

3.4.3 Lower Bounds on the Excess Edges

In this section, we provide some lower bounds on the number of excess edges required to grow a graph if we fix the number of slots to $\log n$. For simplicity, we assume that $n = 2^\delta$ for some integer δ, but this assumption can be dropped.

We define a particular graph G_{min} of size n, through a growth schedule σ_{min} for it. The schedule σ_{min} contains $\log n$ slots. In every slot t, the schedule generates one vertex u' for every vertex u in $(G_{min})_{t-1}$ and activates uu'. This completes the description of σ_{min}. Let G be any graph on n vertices, grown by a $\log n$-slot schedule σ. Observe that any edge activated by σ_{min} is also activated by σ. Thus, any edges of G_{min} "not used" by G are excess edges that must be deleted by σ, for G to be grown by it. The latter is captured by the following minimum edge-difference over all permutations of $V(G)$ mapped on $V(G_{min})$.

Consider the set B of all possible bijections between the vertex sets of $V(G)$ and $V(G_{min})$, $b : V(G) \longmapsto V(G_{min})$. We define the edge-difference ED_b of every such bijection $b \in B$ as $ED_b = |\{uv \in E(G_{min}) \mid b(u)b(v) \notin E(G)\}|$. The minimum edge-difference over all bijections $b \in B$ is $\min_b ED_b$. We argue that a growth schedule of $\log n$ slots for graph G uses at least $\min_p ED_p$ excess edges since the schedule has to activate every edge of G_{min} and then delete at least the minimum edge-difference to get G. This property leads to the following theorem, which can then be used to obtain lower bounds for specific graph families.

Theorem 3.34. *Any growth schedule σ of $\log n$ slots that grows a graph G of n vertices, uses at least $\min_b ED_b$ excess edges.*

Proof. Since every schedule σ of $\log n$ slots activates every edge uv of G_{min_i}, σ must delete every edge $uv \notin G$. To find the minimum number of such edges, if we consider the set B of all possible bijections between the vertex sets of $V(G)$ and $V(G_{min})$, $b : V(G) \longmapsto V(G_{min})$ and we compute the minimum edge-difference over all bijections $b \in B$ as $\min_b ED_b$, then schedule σ has to activate every edge of G_{min} and delete at least $\min_b ED_b$ edges. □

Corollary 3.35. *Any growth schedule of* $\log n$ *slots that grows a path or star graph of* n *vertices, uses* $\Omega(n)$ *excess edges.*

Proof. Note that for a star graph $G = (V, E)$, the maximum degree of a vertex in G_{min} is $\log n$ and the star graph has a center vertex with degree $n-1$. This implies that there are $n-1-\log n$ edges of G_{min} which are not in E. Therefore $\min_b ED_b = (n-1-\log n)$. A similar argument works for the the schedule of a path graph. □

We now define a particular graph $G_{full} = (V, E)$ by providing a growth schedule for it. The schedule contains $\log n$ slots. In every slot t, the schedule generates one vertex u' for every vertex u in G_{t-1} and activates uu'. Upon generating vertex u', it activates an edge $u'v$ with every vertex v that is within $d = 2$ from u'. Assume that we name the vertices $u_1, u_2, \ldots, u_n$, where vertex u_1 was the initiator and vertex u_j was generated in slot $\lceil \log(u_j) \rceil$ and connected with vertex $u_{j-\lceil \log(u_j) \rceil}$.

Lemma 3.36. *If* n *is the number of vertices of* $G_{full} = (V, E)$ *then the number of edges of* G_{full} *is* $n \log n \leq |E| \leq 2n \log n$.

Proof. Let $f(x)$ be the sum of degrees when x vertices have been generated. Clearly $f(2) = 2$. Now consider slot t and lets assume it has x vertices at its end. At end of next slot we have $2x$ vertices. Let the degrees of the vertices at end of slot t be $d_1, d_2, ..., d_k$. Consider now that:

- Child i' of vertex i (generated in slot $t+1$) has 1 edge with its parent and d_i edges (since an edge between it and all vertices at distance 1 from i will be activated in slot t. So $d'_i = d_i + 1$.
- Vertex i has 1 edge (with its child) and d_i edges (one from each new child of its neighbours in slot t), that is $d_i(new) = 2d_i + 1$.

Therefore $f(2x) = 3f(x) + 2x$. Notice that $2f(x) + 2x \leq f(2x) \leq 4f(x) + 4x$. Let $g(x)$ be such that $g(2) = 2$ and $g(2x) = 2g(x) + 2x$. We claim $g(x) = x \log x$. Indeed $g(2) = 2 \log 2 = 2$ and by induction $g(2x) = 2g(x) + 2x = 2x \log x + 2x = 2x \log(2x)$. It follows that $n \log n \leq f(n) \leq 2n \log n$. □

We will now describe the following bipartite graph $G_{bipart} = (V, E)$ using $G_{full} = (V', E')$ to describe the edges of G_{bipart}. Both parts of the graph have $n/2$ vertices and the left part, called A, contains vertices $a_1, a_2, \ldots, a_{n/2}$, and the right part, called B, contains vertices $b_1, b_2, \ldots, b_{n/2}$, and $E' = \{a_i b_j \mid (u_i, u_j \in E) \vee (i = j)\}$. This means that if graph G_{full} has m edges, G_{bipart} has $\Theta(m)$ edges as well.

Theorem 3.37. *Consider graph* $G_{bipart} = (V', E')$ *of size* n. *Any growth schedule* σ *for graph* G_{bipart} *of* $\log n$ *slots uses* $\Omega(n \log n)$ *excess edges.*

Proof. Assume that schedule σ of $\log n$ slots, grows graph G_{bipart}. Since σ has $\log n$ slots, for every vertex $u \in V'_{j-1}$ a vertex must be generated in every slot j in order for the graph to have size n. This implies that in the last slot, $n/2$ vertices have to be generated and we remind that these vertices must be an independent set in G_{bipart}. For $i = \{n/2\}$, $a_i, b_i \in E'$ and both vertices a_i, b_i cannot be generated together in the last slot. This implies that in the last slot, for every $i = 1, 2, \ldots, n/2$, we must have exactly one vertex from each pair of a_i, b_i. Note though that vertices a_1, b_1 have an edge with every vertex in B, A respectively. If vertex a_1 or b_1 is generated in the last slot, only vertices from A or B, respectively, can be generated in that same slot. Thus, we can decide that the last slot must either contain every vertex in A or every vertex in B.

W.l.o.g., assume that in the last slot, we generate every vertex in B. This means that for every vertex $a_i \in A$ one vertex $b_j \in B$ must be generated. Consider an arbitrary vertex a_i for which an arbitrary vertex b_j is generated. In order for this to happen in the last slot, for every $a_l, b_j \in (E' \setminus a_i, b_j)$, $a_l a_i$ must be active and every edge $a_l a_i$ is an excess edge since set A is an independent set in graph G_{bipart}. This means that for each vertex b_j generation, any growth schedule must activate at least $deg(b_j) - 1$ excess edges. By construction, graph G_{bipart} has $O(n \log n)$ edges and thus, the sum of the degrees of vertices in B is $O(n \log n)$. Therefore, any growth schedule has to activate $\Omega(n \log n) - n = \Omega(n \log n)$ excess edges. □

3.5 Conclusion

In this work, we considered a new model for highly dynamic networks, called growing graphs. The model, with no limitation to the edge-activation distance d, allows any target graph G to be constructed, starting from an initial singleton graph, but large values of d are an impractical assumption with simple solutions and therefore we focused on cases where $d = 2$. We defined performance measures to quantify the speed (slots) and efficiency (excess edges) of the growth process, and we noticed that there is a natural trade off between the two. We proposed algorithms for general graph classes that try to balance speed and efficiency. If someone wants super efficient growth schedules (zero excess edges), it is impossible to even find a $n^{\frac{1}{3}-\varepsilon}$-approximation of the length of such a schedule, unless P = NP. For the special case of schedules of $\log n$ slots and $\ell = 0$ excess edges, we provide a polynomial-time algorithm that can find such a schedule.

We believe that the present study, apart from opening new avenues of algorithmic research in graph-generation processes, can inspire work on more applied models of dynamic networks and network deployment, including ones in which the growth process is decentralized and exclusively controlled by the individual network processors and models whose the dynamics are constrained by geometry.

There is a number of interesting technical questions left open by the findings of this paper. It would be interesting to see whether there exists an algorithm that can decide the minimum number of edges required for any schedule to grow a graph G or whether the problem is NP-hard. Note that this problem is equivalent to the cop-win completion problem; that is, ℓ is in this case equal to the smallest number of edges that need to be added to G to make it a cop-win graph. We mostly focused on the two extremes of the (k, ℓ)-spectrum, namely one in which k is close to $\log n$ and the other is which ℓ close to zero. The in-between landscape remains to be explored. Finally, we gave some efficient algorithms, mostly for specific graph families, but there seems to be room for more positive results. It would also be interesting to study a combination of the growth dynamics of the present work and the edge-modification dynamics of [79], thus, allowing the activation of edges between vertices generated in past slots.

Chapter 4

Programmable Matter

4.1 Introduction

The programmable matter field is quite a new area in the field of theoretical computer science. Apart from the fact that systems work is still in its infancy, there is also an apparent lack of unifying formalism and theoretical treatment. Still there are some first theoretical efforts aiming at understanding the fundamental possibilities and limitations of this prospective. The area of *algorithmic self-assembly* tries to understand how to program molecules (mainly DNA strands) to manipulate themselves, grow into machines and at the same time control their own growth [11]. The theoretical model guiding the study in algorithmic self-assembly is the Abstract Tile Assembly Model (aTAM) [41, 90] and variations. A model called the *nubot* model, was proposed for studying the complexity of self-assembled structures with active molecular components [14]. This model "is inspired by biology's fantastic ability to assemble biomolecules that form systems with complicated structure and dynamics, from molecular motors that walk on rigid tracks and proteins that dynamically alter the structure of the cell during mitosis, to embryonic development where large-scale complicated organisms efficiently grow from a single cell" [14]. Another model, called the *Network Constructors* model, studied what stable networks can be constructed by a population of finite-automata that interact randomly like molecules in a well-mixed solution and can establish bonds with each other according to the rules of a common small protocol [23]. The development of Network Constructors was based on the *Population Protocol* model of Angluin *et al.* [3], that does not include the capability of creating bonds and focuses more on the computation of functions on inputs. A very interesting fact about population protocols is that, when operating under a uniform random scheduler, they are formally equivalent to a restricted version of stochastic *chemical reaction networks* (CRNs), "which model chemistry in a *well-mixed solution* and are widely used to describe information processing occurring in natural cellular regulatory networks" (see, e.g., [91, 92]). Also the

proposed *Amoebot* model, "offers a versatile framework to model self-organizing particles and facilitates rigorous algorithmic research in the area of programmable matter" [10, 93–96]. See also [97] for latest activities and developments in this area of research.

Each theoretical approach, and to be more precise, each individual model, has its own beauty and has lead to different insights and developments regarding potential programmable matter systems of the future and in some cases to very intriguing technical problems and open questions. Still, it seems that the right way for theory to boost the development of more refined real systems is to reveal the *transformation capabilities of mechanisms and technologies that are available now*, rather than by exploring the unlimited variety of theoretical models that are not expected to correspond to a real implementation in the near future.

In this chapter, we follow such an approach, by studying the transformation capabilities of models for programmable matter, which are based on minimal mechanical capabilities, easily implementable by existing technology.

4.1.1 Our Approach

We study a minimal programmable matter system consisting of n cycle-shaped modules, with each module (or *node*) occupying at any given time a cell of the 2D grid (no two nodes can occupy the same cell at the same time). Therefore, the composition of the programmable matter systems under consideration is discrete. Our main question throughout is whether an initial arrangement of the material can *transform* (either in principle, e.g., by an external authority, or by itself) to some other target arrangement. In more technical terms, we are provided with an *initial shape* A and a *target shape* B and we are asked whether A *can be transformed to* B via a sequence of *valid* transformation steps. Usually, a step consists either of a *valid movement* of a single node (in the *sequential case*) or of more than one node at the same time (in the *parallel case*). We consider two quite primitive types of movement. The first one, called *rotation*, allows a node to rotate 90 degrees around one of its neighbors either clockwise or counterclockwise (see, e.g., Figure 4.1 in Section 4.2) and the second one, called *sliding*, allows a node to slide by one position "over" two neighboring nodes (see, e.g., Figure 4.2 in Section 4.2). Both movements succeed only if the whole direction of movement is free of obstacles (i.e., other nodes blocking the way). More formal definitions are provided in Section 4.2. One part of the chapter focuses on the case in which only rotation is available to the nodes and the other part studies the case in which both rotation and sliding are available. The latter case has been studied to some extent in the past in the, so called, *metamorphic systems* [13, 42, 43], which makes those studies the closest to our approach.

For rotation only, we introduce the notion of *color-consistency* and prove that if two shapes are not color-consistent then they cannot be transformed to each other.

On the other hand color-consistency does not guarantee transformability, as there is an infinite set of pairs (A, B) such that A and B are color consistent but still cannot be transformed to each other. At this point, observe that if A can be transformed to B then the inverse is also true, as all movements considered in this chapter are *reversible*. We distinguish two main types of transformations: those that are allowed to break the connectivity of the shape during the transformation and those that are not; we call the corresponding problems Rot-Transformability and RotC-Transformability, respectively. We prove that RotC-Transformability is a proper subset of Rot-Transformability by showing that a line-folding problem is in Rot-Transformability\RotC-Transformability. Our main result regarding Rot-Transformability is that Rot-Transformability $\in \mathbf{P}$. To prove polynomial-time decidability, we prove that two shapes A and B are transformable to each other iff both A and B have at least one movement available (without any movement available, a shape is *blocked* and can only trivially transform to itself). Therefore, transformability reduces to checking the availability of a movement in the initial and target shapes. The idea is that if a movement is available in a shape A, then there is always a way to *extract* from A a *2-line* (meaning to disconnect a line of length 2 from shape A, via a sequence of rotation movements). Such a 2-line can move freely in any direction and can also extract further nodes to form a *4-line*. A 4-line in turn can also move freely in any direction and is also capable of extracting nodes from the shape and transferring them, one at a time, to any desired target position. In this manner, the 4-line can transform A into a line with leaves around it that is color-consistent to A (based on a proposition that we prove, stating that any shape has a corresponding color-consistent line-with-leaves). Similarly, B, given that it is color-consistent with A, can be transformed by the same approach to exactly the same line-with-leaves, and then, by reversibility, it follows that A and B can be transformed to each other by using the line-with-leaves as an intermediate. This set of transformations do not guarantee the preservation of connectivity during the transformation. That is, even though the initial and target shapes considered are connected shapes, the shapes formed at intermediate steps of the transformation may very well be disconnected shapes.

We next study RotC-Transformability, in which again the only available movement is rotation, but now connectivity of the material has to be preserved throughout the transformation. The property of preserving the connectivity is expected to be a crucial property for programmable matter systems, as it allows the material to maintain coherence and strength, to eliminate the need for wireless communication, and, finally, enables the development of more effective power supply schemes, in which the modules can share resources or in which the modules have no batteries but are instead constantly supplied with energy by a centralized source (or by a supernode that is part of the material itself). Such benefits can lead to simplified designs and potentially to

reduced size of individual modules. We first prove that RotC-Transformability $\in$ **PSPACE**. The rest of our results here are strongly based on the notion of a *seed*. This stems from the observation that a large set of infeasible transformations become feasible by introducing to the initial shape an additional, and usually quite small, seed; i.e., a small shape that is being attached to some point of the initial shape in order to trigger an otherwise infeasible transformation. In particular, we prove that a *3-line seed*, if placed appropriately, is sufficient to achieve folding of a line (otherwise impossible). We then investigate seeds that could serve as components capable of traveling the perimeter of an arbitrary connected shape A. Such seed-shapes are very convenient as they are capable of "simulating" the *universal transformation* techniques that are possible if we have both rotation and sliding movements available (discussed in the next paragraph). To this end, we prove that all seeds of size ≤ 4 cannot serve for this purpose, by proving that they cannot even walk the perimeter of a simple line shape. Then we focus on a *6-seed* and prove that such a seed is capable of walking the perimeter of a large family of shapes, called *orthogonally convex* shapes. This is a first indication, that there might be a large family of shapes that can be transformed to each other with rotation only and without breaking connectivity, by extracting a 6-seed and then exploiting it to transfer nodes to the desired positions. To further support this, we prove that the 6-seed is capable of performing such transfers, by detaching pairs of nodes from the shape, attaching them to itself, thus forming an *8-seed* and then being still capable to walk the perimeter of the shape.

Next, we consider the case in which both rotation and sliding are available and insist on connectivity preservation. We first provide a proof that this combination of simple movements is *universal* with respect to (abbreviated "w.r.t." throughout) transformations, as any pair of connected shapes A and B of the same order ("order" of a shape S meaning the number of nodes of S throughout the chapter) can be transformed to each other without ever breaking the connectivity throughout the transformation (a first proof of this fact had already appeared in [43]). This generic transformation requires $\Theta(n^2)$ sequential movements in the worst case. By a potential-function argument we show that no transformation can improve on this worst-case complexity for some specific pairs of shapes and this lower bound is independent of connectivity preservation; it only depends on the inherent *transformation-distance* between the shapes. To improve on this, either some sort of parallelism must be employed or more powerful movement mechanisms, e.g., movements of whole sub-shapes in one step. We investigate the former approach and prove that there is a *pipelining* general transformation strategy that improves the time to $O(n)$ (parallel time). We also give a matching $\Omega(n)$ lower bound. On the way, we also show that this parallel complexity is feasible even if the nodes are labeled, meaning that individual nodes must end up in specific positions of the target-shape.

Finally, we assume that the nodes are distributed processes equipped with limited local memory and able to perform communicate-compute-move rounds (where, again, both rotation and sliding movements are available) and provide distributed algorithms for a general type of transformation.

In Section 4.1.2 we discuss further related literature. Section 4.2 brings together all definitions and basic facts that are used throughout the chapter. In Section 4.3, we study programmable matter systems equipped only with rotation movement. In Section 4.4, we insist on rotation only, but additionally require that the material maintains connectivity throughout the transformation. In Section 4.5, we investigate the combined effect of rotation and sliding movements.

4.1.2 Further Related Work

Mobile and Reconfigurable Robotics. There is a very rich literature on mobile and reconfigurable robotics. In mobile (swarm) robotics systems and models, as are, for example, the models for robot gathering [98, 99] and deployment [100] (cf., also [101]), geometric pattern formation [102, 103], and connectivity preservation [104], the modules are usually robots equipped with some mobility mechanism making them free to move in any direction of the plane (and in some cases even continuously). In contrast, we only allow discrete movements relative to neighboring nodes. Modular self-reconfigurable robotic systems form an area on their own, focusing on aspects like the design, motion planning, and control of autonomous robotic modules [105–108]. The model considered in this chapter bears similarities to some of the models that have appeared in this area. The main difference is that we follow a more computation-theoretic approach, while the studies in this area usually follow a more applied perspective. Since the research of this chapter has been published, a lot of follow-up work has been considered where the authors extend the model with more powerful movement [109, 110].

Puzzles. Puzzles are combinatorial one-player games, usually played on some sort of board. Typical questions of interest are whether a given puzzle is solvable and finding the solution that makes the fewest number of moves. Answers to these questions range from being in **P** up to **PSPACE**-hard or even undecidable when some puzzles are generalized to the entire plane with unboundedly many pieces [111, 112]. Famous examples of puzzles are the Fifteen Puzzle, Sliding Blocks, Rush Hour, Pushing Blocks, and Solitaire. Even though none of these is equivalent to the model considered here, the techniques that have been developed for solving and characterizing puzzles may turn out to be very useful in the context of programmable matter systems. Actually, in some cases, such puzzles show up as special cases of the transformation problems considered here (e.g., the Fifteen Puzzle may be obtained if we restrict a transformation of node-labeled shapes to take place in a 4x4 square region); such connections of

programmable matter systems and models to puzzles have also been highlighted in [113].

Metamorphic Systems. As already mentioned, [13, 43] are very close to our approach therefore, despite the fact that they naturally belong to models of mobile and reconfigurable robotics, we discuss them separately. All modeling and results in the present chapter were developed independently from the above studies, but as there are some similarities we now compare to those papers. In particular, the model of rotations and slidings on the grid that we studied, turns out to be the same as the model considered in those papers. The study of rotation alone that we explore in Sections 4.3 and 4.4 to the best of our knowledge has not been considered in the literature. Regarding rotation and sliding combined, discussed in Section 4.5, the universality result that we prove turns out to have been first proven in [43], but we leave our proof for completeness. Both proofs exploit the idea of converting a given shape A first into a straight line and then to the target shape B, but the proof arguments are different. Regarding [42], in that paper the authors mainly studied a distributed version of the transformation problems. They also posed a number of decidability questions, some of them proved to be undecidable (we do not deal with undecidable problems in this chapter). Their main question regarding decidability problems was related to the universality result later proved in [43]. Our distributed results, only briefly mentioned at the end of Section 4.5 (to be published separately, as the focus of the present chapter is feasibility of transformations and centralized transformation algorithms), use a different model than the one studied in [13]. The main difference is that in [13] each node can communicate to the whole network in any given round (mostly, though, in the sense that a node can have a complete observation of the current global configuration), while in our case communication is restricted to be local. An interesting idea, explored in [42] that has also been exploited in our study (but for centralized transformations in our case) is the pipelining technique, by which more than one nodes traverse the perimeter of a shape at the same time in order to speed up transformation. In their case, it is used on the special case of connected shapes in which no row has a gap, i.e., every row is a line of consecutive nodes, while in our case it is used to speed up the universality result to linear parallel time. Finally, it should be mentioned that some authors have studied alternative geometries than the one considered here, such as representations of systems in which each module is a regular hexagon (see, for instance, [114, 115].

4.2 Preliminaries

The programmable matter systems considered in this chapter operate on a 2D square grid, with each position (or *cell*) of the grid being uniquely referred to by its $y \geq 0$

and $x \geq 0$ coordinates. [1] Such a system consists of a set V of n *modules*, called *nodes* throughout. Each node may be viewed as a spherical module fitting inside a cell of the grid. At any given time, each node $u \in V$ occupies a cell $o(u) = (o_y(u), o_x(u)) = (i, j)$ (omitting the time index for simplicity here and also whenever clear from context) and no two nodes may occupy the same cell. [2] In some cases, when a cell is occupied by a node we may refer to that cell by a color, e.g., *black*, and when a cell is not occupied (in which case we may also call it *empty*) we usually color it white. At any given time t, the positioning of nodes on the grid defines an undirected *neighboring relation* $E(t) \subset V \times V$, where $\{u, v\} \in E$ iff $o_y(u) = o_y(v)$ and $|o_x(u) - o_x(v)| = 1$ or $o_x(u) = o_x(v)$ and $|o_y(u) - o_y(v)| = 1$, that is, if u and v are either *horizontal* or *vertical* neighbors on the grid, respectively. It is immediate to observe that every node can have at most 4 neighbors at any given time. A more informative way to define the system at a given time t, and thus often more convenient, is as a mapping $P_t \colon \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \to \{0, 1\}$ where $P_t(i, j) = 1$ iff cell (i, j) is occupied by a node.

At any given time t, $P_t^{-1}(1)$ defines a *shape*. [3] Such a shape is called *connected* if $E(t)$ defines a connected graph. A connected shape is called *convex* if for any two occupied cells, the line in $\mathbb{R}^2$ through their centers does not pass through an empty cell. A connected shape A is called *orthogonally convex* if the intersection of any vertical or horizontal line with A is either empty or connected. Equivalently, this means that for any two cells occupied by A, belonging either to the same row or the same column, the line that connects their centers does not pass through an empty cell (i.e., compared to pure convexity, we now exclude diagonal lines). We call a shape *compact* if it has no holes.

In general, shapes can *transform* to other shapes via a sequence of one or more *movements* of individual nodes. Time consists of discrete *steps* (or *rounds*) and in every step, zero or more movements may occur, possibly following a centralized or distributed computation sub-step, depending on the application. In the *sequential* case, at most one movement may occur per step, and in the *parallel* case any number of "valid" movements may occur in parallel. [4] We consider two types of movements: (i) *rotation* and (ii) *sliding*. In both movements, a single node moves relative to one or more neighboring nodes as we now explain.

[1] We should make clear at this point that the grid assumption is only used to facilitate intuition of the geometric properties of the model (that is, the permissible arrangements of the nodes) and presentation of our results. Therefore, the systems in this chapter consist of the nodes only and the grid is not considered part of the studied systems.

[2] As these are discrete coordinates, we have preferred to use the (i, j) matrix notation, where i is the row and j the column, instead of the (x, y) Cartesian coordinates, but this doesn't make any difference.

[3] P_t^{-1} denotes the inverse function of P_t, therefore $P_t^{-1}(1)$ returns the set of all cells that are occupied by nodes.

[4] By "valid", we mean here subject to the constraint that their whole movement paths correspond to pairwise disjoint sub-areas of the grid.

A single *rotation* movement of a node u is a 90 degrees *rotation* of u around one of its neighbors. Let (i, j) be the current position of u and let its neighbor be v occupying the cell $(i-1, j)$ (i.e., lying below u). Then u can *rotate* 90 degrees clockwise (counterclockwise) around v iff the cells $(i, j+1)$ and $(i-1, j+1)$ ($(i, j-1)$ and $(i-1, j-1)$, respectively) are both empty. By rotating the whole system 90 180 and 270 all possible rotation movements are defined analogously. See Figure 4.1 for an illustration.

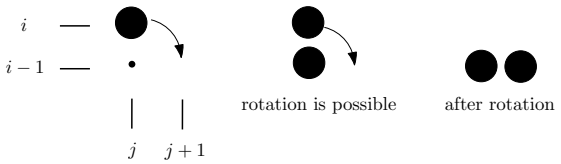

FIGURE 4.1: Rotation clockwise. A node on the black dot (in row $i-1$) and empty cells at positions $(i, j+1)$ and $(i-1, j+1)$ are required for this movement. Then an example movement is given.

A single *sliding* movement of a node u is a one-step horizontal or vertical movement "over" a horizontal or vertical line of (neighboring) nodes of length 2. In particular, if (i, j) is the current position of u, then u can *slide* rightwards to position $(i, j+1)$ iff $(i, j+1)$ is not occupied and there exist nodes at positions $(i-1, j)$ and $(i-1, j+1)$ or at positions $(i+1, j)$ and $(i+1, j+1)$, or both. Precisely the same definition holds for up, left, and down sliding movements by rotating the whole system 90 , 180, and 270 degrees counterclockwise, respectively. Intuitively, a node can slide one step in one direction if there are two consecutive nodes either immediately "below" or immediately "above" that direction that can assist the node slide (see Figure 4.2). [5]

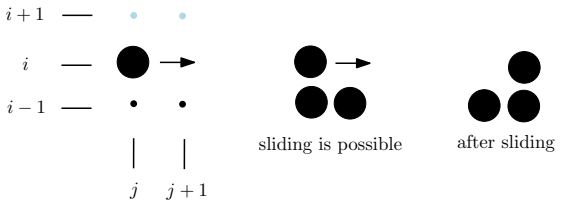

FIGURE 4.2: Sliding to the right. Either the two light blues (dots in row $i+1$; appearing gray in print) or the two blacks (dots in row $i-1$) and an empty cell at position $(i, j+1)$ are required for this movement. Then an example movement with the two blacks is given.

Let A and B be two shapes. We say that A *transforms to* B *via a movement* m (which can be either a rotation or a sliding), denoted $A \xrightarrow{m} B$, if there is a node u in A such that if u applies m, then the shape resulting after the movement is B (possibly

[5] Observe that there are plausible variants of the present definition of sliding, such as to slide with nodes at $(i-1, j)$ and $(i+1, j+1)$ or even with a single node at $(i-1, j)$ or at $(i+1, j)$. In this chapter, though, we only focus on our original definition.

after rotations and translations of the resulting shape, depending on the application). We say that A *transforms in one step to* B (or that B *is reachable in one step from* A), denoted $A \to B$, if $A \xrightarrow{m} B$ for some movement m. We say that A *transforms* to B (or that B is *reachable* from A) and write $A \rightsquigarrow B$, if there is a sequence of shapes $A = C_0, C_1, \ldots, C_t = B$, such that $C_i \to C_{i+1}$ for all i, $0 \leq i < t$. We should mention that we do not always allow m to be any of the two possible movements. In particular, in Sections 4.3 and 4.4 we only allow m to be a rotation. We shall clearly explain what movements are permitted in each part of the chapter. Observe now that both rotation and sliding are *reversible* movements, a fact that we extensively use in our results. Based on this, we may prove that:

Proposition 4.1. *The relation "transforms to" (i.e., '$\rightsquigarrow$') is a partial equivalence relation.*

Proof. The relation '$\rightsquigarrow$' is a binary relation on shapes. To show that it is a partial equivalence relation, we have to show that it is symmetric and transitive.

For symmetry, we have to show that for all shapes A and B, if $A \rightsquigarrow B$ then $B \rightsquigarrow A$. It suffices to show that for all A, B, if $A \to B$ then $B \to A$, meaning that every one-step transformation (which can be either a single rotation or a single sliding) can be *reversed*. For the rotation case, this follows by observing that a rotation of a node u can be performed iff there are two consecutive empty positions in its trajectory. When u rotates, it leaves its previous position empty, thus, leaving in this way two consecutive positions empty for the reverse rotation to become enabled. The argument for sliding is similar.

For transitivity, we have to show that for all shapes A, B, and C, if $A \rightsquigarrow B$ and $B \rightsquigarrow C$ then $A \rightsquigarrow C$. By definition, $A \rightsquigarrow B$ if there is a sequence of shapes $A = C_0, C_1, \ldots, C_t = B$, such that $C_i \to C_{i+1}$ for all i, $0 \leq i < t$ and $B \rightsquigarrow C$ if there is a sequence of shapes $B = C_t, C_{t+1}, \ldots, C_{t+l} = C$, such that $C_i \to C_{i+1}$ for all i, $t \leq i < t + l$. So, for the sequence $A = C_0, C_1, \ldots, C_t = B, C_{t+1}, \ldots, C_{t+l} = C$ it holds that $C_i \to C_{i+1}$ for all i, $0 \leq i < t + l$, that is, $A \rightsquigarrow C$. □

When the only available movement is rotation, there are shapes in which no rotation can be performed (such examples are provided in Section 4.3). If we introduce a *null* rotation, then every shape may transform to itself by applying the *null* rotation. That is, reflexivity is also satisfied, and, together with symmetry and transitivity from Proposition 4.1, '$\rightsquigarrow$' (by rotations only) becomes an equivalence relation.

Definition 4.2. Let A be a connected shape. Color black each cell of the grid that is occupied by a node of A. A cell (i, j) is part of a *hole* of A if every infinite length single path starting from (i, j) (moving only horizontally and vertically) necessarily goes through a black cell. Color black also every cell that is part of a hole of A, to

obtain a compact black shape A'. Consider now polygons defined by unit-length line segments of the grid. Define the *perimeter* of A as the minimum-area such polygon that completely encloses A' in its interior. The fact that the polygon must have an *interior* and an *exterior* follows directly from the Jordan curve theorem [116].

Definition 4.3. Now, color red any cell of the grid that has contributed at least one of its line segments to the perimeter and is not black (i.e., is not occupied by a node of A). Call this the *cell-perimeter* of shape A. See Figure 4.3 for an example.

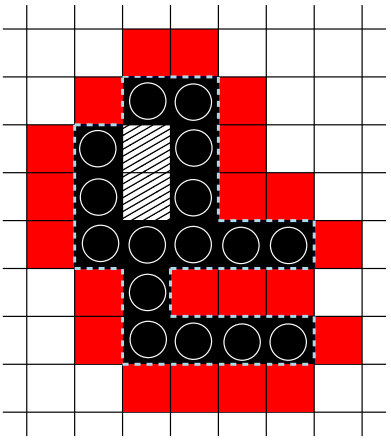

FIGURE 4.3: The perimeter (polygon of unit-length dashed line segments colored light blue; appearing dashed gray in print) and the cell-perimeter (cells colored red; appearing gray in print, throughout the chapter) of a shape A (white spherical nodes; their corresponding cells have been colored black). The dashed black cells correspond to a hole of A.

Definition 4.4. The *external surface* of a connected shape A, is a shape B, not necessarily connected, consisting of all nodes $u \in A$ such that u occupies a cell defining at least one of the line segments of A's perimeter.

Definition 4.5. The *extended external surface* of a connected shape A, is defined by adding to A's external surface all nodes of A whose cell shares a corner with A's perimeter (for example, the black node just below the hole in Figure 4.3).

Proposition 4.6. *The extended external surface of a connected shape A, is itself a connected shape.*

Proof. The perimeter of A is connected, actually, it is a cycle. This connectivity is preserved by the extended external surface, as whenever the perimeter moves straight, we have two horizontally or vertically neighboring nodes on the extended external surface and whenever it makes a turn, we either stay put or preserve connectivity via an intermediate diagonal node (from those nodes used to extend the external surface). □

Observe, though, that the extended external surface is not necessarily a cycle. For example, the extended external surface of a line-shape is equal to the shape itself (and, therefore, a line).

4.2.1 Problem Definitions

We here provide formal definitions of all the transformation problems that are considered in this chapter.

ROT-TRANSFORMABILITY. Given a connected initial shape A and a connected target shape B, decide whether A can be transformed to B (under translations and rotations of the shapes) using only a sequence of rotation movements.

ROTC-TRANSFORMABILITY. The special case of ROT-TRANSFORMABILITY in which A and B are again connected shapes and connectivity must be preserved throughout the transformation.

RS-TRANSFORMABILITY. Given a connected initial shape A and a connected target shape B, decide whether A can be transformed to B (under translations and rotations of the shapes) by a sequence of rotation and sliding movements.

Minimum-Seed-Determination. Given an initial shape A and a target shape B (usually only with rotation available and a proof that A and B are not transformable to each other without additional assumptions) determine a minimum-size seed and an initial positioning of that seed relative to A that makes the transformation from A to B feasible. There are several meaningful variations of this problem. For example, the seed may or may not be eventually part of the target shape or the seed may be used as an intermediate step to show feasibility with "external" help and then be able to show that, instead of externally providing it, it is possible to *extract* it from the initial shape A via a sequence of moves. We will clearly indicate which version is considered in each case.

In the above problems, the goal is to show feasibility of a set of transformation instances and, if possible, to provide an algorithm that decides feasibility. [6]

In the last part of the chapter, we consider *distributed transformation tasks*. There, the nodes are *distributed processes* able to perform *communicate-compute-move rounds* and the goal is to program them so that they (algorithmically) self-transform their

[6] An immediate next goal is to devise an algorithm able to compute an actual transformation or even compute or approximate the *optimum* transformation (usually w.r.t. the number of moves). We leave these as interesting open problems.

initial arrangement to a target arrangement.

Distributed-Transformability. Given an initial shape A and a target shape B (usually by having access to both rotation and sliding), the nodes (which are now distributed processes), starting from A, must transform themselves to B by a sequence of communication-computation-movement rounds. In the distributed transformations, we mostly consider the case in which A can be *any connected shape* and B is a *spanning line*, i.e., a linear arrangement of all the nodes.

4.3 Rotation

In this section, the only permitted movement is 90*rotation* around a neighbor. Our main result in this section is that ROT-TRANSFORMABILITY $\in \mathbf{P}$.

Consider a black and red checkered coloring of the 2D grid, similar to the coloring of a chessboard. Then any shape S may be viewed as a colored shape consisting of $b(S)$ blacks and $r(S)$ reds ($b(S)$ and $r(S)$ denoting the number of black and reds, respectively, of a shape S). Call two shapes A and B *color-consistent* if $b(A) = b(B)$ and $r(A) = r(B)$ and call them *color-inconsistent* otherwise. Call a transformation from a shape A to a shape C *color-preserving* if A and C are color consistent. Observe now, that if $A \to B$, then A and B are color-consistent, because a rotation can never move a node to a position of different color than its starting position. This implies that if $A \rightsquigarrow C$, then A and C are color-consistent, because any two consecutive shapes in the sequence are color-consistent. We conclude that:

Observation 7. The rotation movement is color-preserving. Formally, $A \rightsquigarrow C$ (restricted to rotation only) implies that A and C are color-consistent. In particular, every node beginning from a black (red) position of the grid, will always be on black (red, respectively) positions throughout a transformation consisting only of rotations.

Based on this property of the rotation movement, we may call each node *black* or *red* throughout a transformation, based only on its initial coloring. Observation 7 gives a partial way to determine that two shapes A and B cannot be transformed to each other by rotations.

Proposition 4.7. *If two shapes A and B are color-inconsistent, then it is impossible to transform one to the other by rotations only.*

We now show that two shapes being color-consistent does not necessarily mean that they can be transformed to each other by rotations. We begin with a proposition relating the number of black and red nodes in a connected shape.

Proposition 4.8. *A connected shape with k blacks has at least $\lceil (k-1)/3 \rceil$ and at most $3k+1$ reds.*

Proof. For the upper bound, observe that a black can hold up to 4 distinct reds in its neighborhood, which implies that k blacks can hold up to $4k$ reds in total, even if the blacks were not required to be connected to each other. To satisfy connectivity, every black must share a red with some other black (if a black does not satisfy this, then it cannot be connected to any other black). Any such sharing reduces the number of reds by at least 1. As at least $k-1$ such sharings are required for each black to participate in a sharing, it follows that we cannot avoid a reduction of at least $k-1$ in the number of reds, which leaves us with at most $4k-(k-1) = 3k+1$ reds.

For the lower bound, if we invert the roles of blacks and reds, we have that l reds can hold at most $3l+1$ blacks. So, if k is the number of blacks, it holds that $k \leq 3l+1 \Leftrightarrow l \geq (k-1)/3$ and due to the fact that the number of reds must be an integer, we conclude that for k blacks the number of reds must be at least $\lceil (k-1)/3 \rceil$. □

Proposition 4.9. *There is a generic connected shape, called* line-with-leaves*, that has a color-consistent version for any connected shape. In other words, for k blacks it covers the whole range of reds from $\lceil (k-1)/3 \rceil$ to $3k+1$ reds.*

Proof. Let red be the majority color of A and k be the number of black nodes of A. Consider a line of alternating red and black nodes, starting and ending with a black node, such that all k black nodes are exhausted (see Figure 4.4). To do this, $k-1$ reds are needed in order to alternate blacks and reds on the line. Next, "saturate" every black (i.e. maximize its degree) by adding as many red nodes as it can fit around it (recall that the maximum degree of every node is 4). The resulting saturated shape has k blacks and $3k+1$ reds. This shape covers the $3k+1$ upper bound on the possible number of reds. By removing red leaf-nodes (i.e., of degree 1) one after the other, we can achieve the whole range of numbers of reds, from $k-1$ to $3k+1$ reds. It suffices to restrict attention to the range from k to $3k+1$ reds. Take now any connected shape A and color it in such a way that red is the majority color, that is $l \geq k$, where l is the number of reds and k is the number of blacks (there is always a way to do that). From the upper bound of Proposition 4.8, l can be at most $3k+1$, so we have $k \leq l \leq 3k+1$ for any connected shape A, which falls within the range that the line-with-leaves can represent. Therefore, we conclude that any connected shape A has a color-consistent shape B from the line-with-leaves family. □

FIGURE 4.4: A saturated line-with-leaves shape, in which there are $k = 5$ blacks and $3k+1 = 16$ reds.

Based on this, we now show that the inverse of Proposition 4.7 is not true, that is, it does not hold that any two color-consistent shapes can be transformed to each other by rotations.

Proposition 4.10. *There is an infinite set of pairs (A, B) of connected shapes, such that A and B are color-consistent but cannot be transformed to each other by rotations only.*

Proof. For shape A, take a rhombus as shown in Figure 4.5, consisting of k^2 blacks and $(k+1)^2$ reds, for any $k \geq 2$. In this shape, every black node is "saturated", meaning that it has 4 neighbors, all of them necessarily red. This immediately excludes the blacks from being able to move, as all their neighboring positions are occupied by reds. But the same holds for the reds, as all potential target-positions for a rotation are occupied by reds. Thus, no rotation movement can be applied to any such shape A and A can only be transformed to itself (by *null* rotations). By Proposition 4.9, any such A has a color-consistent shape B from the family of line-with-leaves shapes, such that $B \neq A$ (actually in B several blacks may have degree 3 in contrast to A where all blacks have degree 4). We conclude that A and B are distinct color-consistent shapes which cannot be transformed to each other, and there is an infinite number of such pairs, as the number k^2 of black nodes of A can be made arbitrarily large. □

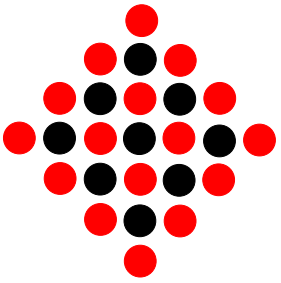

FIGURE 4.5: A rhombus shape, consisting of $k^2 = 9$ blacks and $(k+1)^2 = 16$ reds.

Propositions 4.7 and 4.10 give a partial characterization of pairs of shapes that cannot be transformed to each other. Observe that the impossibilities proved so far hold for all possible transformations based on rotation only, including those that are allowed to break connectivity.

A small shape of particular interest is a bi-color pair or *2-line*. Such pairs can move easily in any direction, which makes them very useful components of transformations. One way to simplify some transformations would be to identify as many such pairs as possible in a shape and treat them in a different way than the rest of the nodes. A question in this respect is whether all the minority-color nodes of a connected shape can be completely matched to (distinct) nodes of the majority color. We show that this is not true.

Proposition 4.11. *There is an infinite family of connected shapes such that if A is a shape of size n in the family, then any matching of A leaves at least $n/8$ nodes of each color unmatched.*

Proof. The counterexample is given in Figure 4.6. The shape consists of a square and a number of flowers emanating from it. A red flower is one with three red petals and a black centre and inversely for black flowers. Observe first that any two consecutive flowers on square cannot be of the same color. We can partition the shape into sub-shapes consisting of 6 nodes of the square and 2 flowers, i.e., 16 nodes in total (such as the top left sub-shape highlighted in Figure 4.6). Then it is sufficient to observe that any at least two petals of any flower cannot get matched to a neighbor of opposite color, therefore 2 red nodes for the red flower and 2 black nodes from the black flower will remain unmatched, which corresponds to $1/8$ of the nodes of the sub-shape will be unmatched reds and another $1/8$ will be unmatched blacks. □

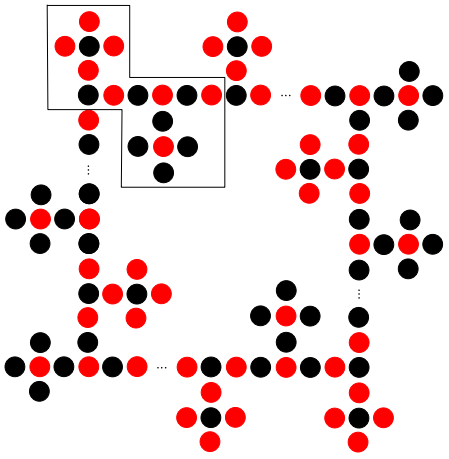

FIGURE 4.6: The counterexample.

Recall that ROT-TRANSFORMABILITY is the language of all transformation problems between connected shapes that can be solved by rotation only and ROTC-TRANSFORMABILITY is its subset obtained by the restriction that the transformation should not break the connectivity of the shape at any point during the transformation. We begin by showing that the inclusion between the two languages is strict, that is, there are strictly more feasible transformations if we allow connectivity to break. We prove this by showing that there is a feasible transformation, namely folding a spanning line in half, in ROT-TRANSFORMABILITY\ROTC-TRANSFORMABILITY.

Theorem 4.12. ROTC-TRANSFORMABILITY $\subset$ ROT-TRANSFORMABILITY.

Proof. ROTC-TRANSFORMABILITY $\subseteq$ ROT-TRANSFORMABILITY is immediate, as any transformation using only rotations that does not break the shape's connectivity is also

a valid transformation for Rot-Transformability. So, it suffices to prove that there is a transformation problem in Rot-Transformability\RotC-Transformability. Consider a (connected) horizontal line of any even length n, and let $u_1, u_2, \ldots, u_n$ be its nodes. The transformation asks to fold the line onto itself, forming a double-line of length $n/2$ and width 2, i.e., a $n/2 \times 2$ rectangle.

It is easy to observe that this problem is not in RotC-Transformability for any $n > 4$: the only nodes that can rotate without breaking connectivity are u_1 and u_n, but any of their two possible rotations only enables a rotation that will bring the nodes back to their original positions. This means that, if the transformation is not allowed to break connectivity, then such a shape is trapped in a loop in which only the endpoints can rotate between three possible positions, therefore it is impossible to fold a line of length greater than 4.

On the other hand, if connectivity can be broken, we can perform the transformation by the following simple procedure, consisting of $n/4$ phases: In the beginning of every phase $i \in \{1, 2, \ldots, \lfloor n/4 \rfloor\}$, pick the nodes u_{2i-1}, u_{2i}, which shall at that point be the two leftmost nodes of the original line. Rotate u_{2i-1} once clockwise, to move above u_{2i}, then u_{2i} three times clockwise to move to the right of u_{2i-1} (the first of these three rotations breaks connectivity and the third restores it), and then rotate u_{2i-1} twice clockwise to move to the right of u_{2i}, then u_{2i} twice clockwise to move to the right of u_{2i-1} and repeat this alternation until the pair that moves to the right meets the previous pair, which will be when u_{2i-1} becomes the left neighbor of u_{2i-2} on the upper line of the rectangle under formation, or, in case $i = 1$, when u_{2i-1} goes above u_n (see Figure 4.7). If $n/4$ is not an integer, then perform a final phase, in which the leftmost node of the original line is rotated once clockwise to move above its right neighbor, and this completes folding. □

This means that allowing the connectivity to break enables more transformations, and this motivates us to start from this simpler case. But we already know from Proposition 4.10 that even in this case an infinite number of pairs of shapes cannot be transformed to each other. Aiming at a general transformation, we ask whether there is some minimal addition to a shape that would allow it to transform. The solution turns out to be as small as a *2-line seed* (a bi-color pair, usually referred to as "2-line" or "2-seed") lying initially somewhere "outside" the boundaries of the shape (e.g., just below the lowest row occupied by the shape).

Based on the above assumptions, we shall now prove that any pair of color-consistent connected shapes A and B can be transformed to each other. Recall from the discussion before Proposition 4.11 that 2-line shapes can move freely in any direction. The idea is to exploit the fact that the 2-line can move freely in any direction and to use it in order to extract from A another 2-line. In this way, a 4-line seed is formed, which can also move

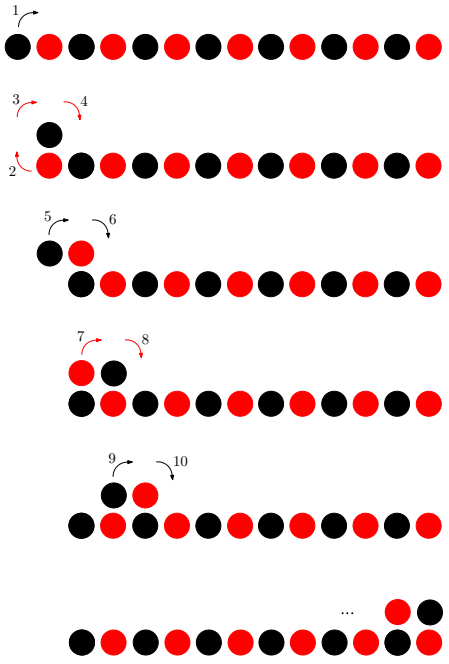

FIGURE 4.7: Line folding.

freely in all directions. Then we use the 4-line as a transportation medium for carrying the nodes of A, one at a time. We exploit these mobility mechanisms to transform A into a uniquely defined shape from the line-with-leaves family of Proposition 4.9. But if any connected shape A with an extra 2-line can be transformed to its color-consistent line-with-leaves version with an extra 2-line, then this also holds inversely due to reversibility, and it follows that any A can be transformed to any B by transforming A to its line-with-leaves version L_A and then inverting the transformation from B to $L_B = L_A$.

Theorem 4.13. *If connectivity can break and there is a 2-line seed provided "outside" the initial shape, then any pair of color-consistent connected shapes A and B can be transformed to each other by rotations only.*

Proof. Without loss of generality (abbreviated "w.l.o.g." throughout), due to symmetry and the 2-line's unrestricted mobility, it suffices to assume that the seed is provided somewhere below the lowest row l occupied by the shape A. We show how A can be transformed to L_A with the help of the seed. We define L_A as follows: Let k be the cardinality of the minority color, let it be the black color. As there are at least k reds, we can create a horizontal line of length $2k$, i.e., $u_1, u_2, \ldots, u_{2k}$, starting with a black (i.e., u_1 is black), and alternating blacks and reds. In this way, the blacks are exhausted. The remaining $\leq (3k+1)-k = 2k+1$ reds are then added as leaves of the black nodes, starting from the position to the left of u_1 and continuing counterclockwise, i.e., below u_1, below u_3, ..., below u_{2k-1}, above u_{2k-1}, above u_{2k-3}, and so on. This gives the same

shape from the line-with-leaves family, for all color-consistent shapes (observe that the leaf to the right of the line is always placed). L_A shall be constructed on rows $l-5$ to $l-3$ (not necessarily inclusive), with u_1 on row $l-4$ and a column j preferably between those that contain A.

First, extract a 2-line from A, from row l, so that the 2-line seed becomes a 4-line seed. To see that this is possible for every shape A of order at least 2, distinguish the following two cases: (i) If the lowest row has a horizontal 2-line, then the 2-line can leave the shape without any help and approach the 2-seed. (ii) If not, then take any node u of row l. As A is connected and has at least two nodes, u must have a neighbor v above it. The only possibility that the 2-line u,v is not free to leave A is when v has both a left and a right neighbor. Figure 4.8 shows how this can be resolved with the help of the 2-line seed (now the 2-line seed approaches and extracts the 2-line).

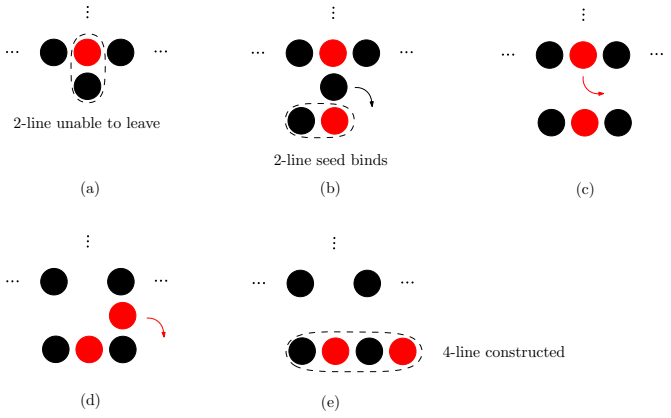

FIGURE 4.8: Extracting a 2-line with the help of the 2-line seed.

To transform A to L_A, given the 4-line seed, do the following:

- While black is still present in A:
  - If on the current lowest row occupied by A, there is a 2-line that can be extracted alone and moved towards L_A, then perform the shortest such movement that attaches the 2-line to the right endpoint of L_A's line $u_1, u_2, \ldots$.
  - If not, then do the following. Maintain a *repository* of nodes at the empty space below row $l-7$, initially empty. If, either in the lowest row of A or in the repository, there is a node of opposite color than the current color of the right endpoint of L_A's line, use the 4-line to transfer such a node and make it the new right endpoint of L_A's line. Otherwise, use the 4-line to transfer a node of the lowest row of A to the repository.
- Once black has been exhausted from A and the repository (i.e., when u_{2k-3} has been placed; u_{2k-1} and u_{2k} will only be placed in the end as they are part of

the 4-line), transfer a red to position u_{2k-2}. If there are no more nodes left, run the termination phase, otherwise transfer the remaining nodes (all red) with the 4-line, one after the other, and attach them as leaves around the blacks of L_A's line, beginning from the position to the left of u_1 counterclockwise, as described above (skipping position u_{2k}).

- Termination phase: the line-with-leaves is ready, apart from positions u_{2k-1}, u_{2k} which require a 2-line from the 4-line. If the position above u_{2k-1} is empty, then extract a 2-line from the 4-line and transfer it to the positions u_{2k-1}, u_{2k}. This completes the transformation. If the position above u_{2k-1} is occupied by a node u_{2k+1}, then place the whole 4-line vertically with its lowest endpoint on u_{2k} (as in Figure 4.9). Then rotate the top endpoint counterclockwise to move above u_{2k+1}, then rotate u_{2k+1} clockwise around it to move to its left, then rotate the node above u_{2k} counterclockwise to move to u_{2k-1}, and finally restore u_{2k+1} to its original position. This completes the construction (the 2-line that always remains can be transferred in the end to a predetermined position).

□

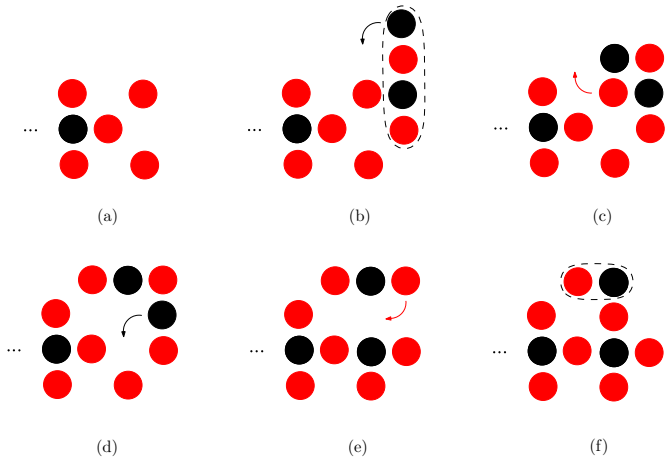

FIGURE 4.9: The termination phase of the transformation.

The natural next question is to what extent the 2-line seed assumption can be dropped. Clearly, by Proposition 4.10, this cannot be always possible. The following corollary gives a sufficient condition to drop the 2-line seed assumption, without looking deep into the structure of the shapes that satisfy it.

Corollary 4.14. *Assume rotations only and that connectivity can break. Let A and B be two color-consistent connected shapes such that each one of them can self-extract a 2-line. Then A and B can be transformed to each other.*

We remind that a rotation move in a grid can occur towards four directions: $North\ East(1)$, $SouthEast(2)$, $SouthWest(3)$, $NorthWest(4)$. In order for the first move to occur, a node has to be present North OR East but not both, and similarly for the other directions. If the connectivity of the shape can be broken and two nodes, u and v, are next to each other and u can perform a rotation using v, then v can perform a rotation using u if the connectivity of the shape can be broken.

We now arrive at a sufficient and necessary condition for dropping the 2-line seed assumption.

Lemma 4.15. *A 2-seed can be extracted from a shape A iff at least one node of A can rotate.*

Proof. If no node of A can rotate, then no movement is possible and trivially a 2-seed cannot be extracted from the shape. Therefore it suffices to prove that if any node of A can rotate then a 2-seed can be extracted from A (meaning that after a sequence of permissible movements a 2-seed will end up at the exterior of A, i.e., outside its perimeter). We distinguish two cases:

- *Case 1: There is a node u incident to the perimeter of A that can move.* In this case, there must be at least two neighboring nodes u, v at positions $(i, j), (i, j+1)$ such that there are no nodes at position $(i + 1, j), (i + 1, j + 1)$ or in another symmetric orientation. In any of these cases, the 2-seed consists of the nodes u, v and can readily be extracted from shape A. The only exception is when the empty space is part of a tunnel of width 1. In this case, by focusing to the nodes adjacent to the outermost two empty cells of the tunnel, these two nodes can be extracted as a 2-seed.

- *Case 2: There is an internal node u (i.e., not incident to the perimeter) of A that can move. In this case, w.l.o.g. (because all other cases are symmetric to the one considered) there must be two consecutive empty cells $(i, j), (i, j + 1)$ with the two cells $(i - 1, j), (i - 1, j + 1)$ below them being occupied by nodes. It is straightforward to reconfigure these four cells by two permissible rotations, so that cells $(i - 1, j), (i - 1, j + 1)$ become now empty and cells $(i, j), (i, j + 1)$ occupied. By induction, as long as the cells below are occupied we can continue shifting the two consecutive empty cells downwards. If we now focus on the downmost 2 such empty cells irrespective of their orientation (the other directions are symmetric)* then shifting them downwards until the perimeter is reached is always possible as the space between those cells and the perimeter is filled with nodes apart possibly from isolated empty cells (i.e., an empty cell surrounded by nodes). In all these cases downward shifting can continue until the perimeter is reached, and when that happens a movement becomes available at the perimeter which is covered by the previous case.

□

Theorem 4.16. ROT-TRANSFORMABILITY $\in \mathbf{P}$.

Proof. In Lemma 4.15, we proved that we can extract a 2-seed from a shape iff a move is initially available. By Theorem 4.13, if both shapes A and B have a 2-seed available then they can be transformed to each other. It follows that two shapes A and B can be transformed to each other iff both have a move available. If yes, *accept*, otherwise, *reject*. These checks can be easily performed in polynomial time as follows. Consider a $n \times n$ grid, in which any shape with n nodes can fit. The time it takes for an algorithm to check if one of the shapes has a move available is $O(n)$. If for example the algorithm checks each individual node, that takes $O(1)$ time and, therefore, $O(n)$ time for n nodes. So for two shapes it takes $O(n)$ time to check if a move is available in each of the shapes. Thus, the problem belongs to $\mathbf{P}$. □

4.4 Rotation and Connectivity Preservation

In this section, we restrict our attention to transformations that transform a connected shape A into one of its color-consistent shapes B using only rotations without ever breaking the connectivity of the shape on the way. As already mentioned in the introduction, connectivity preservation is a very desirable property for programmable matter, as, among other positive implications, it guarantees that communication between all nodes is maintained, it minimizes transformation failures, requires less sophisticated actuation mechanisms, and increases the external forces required to break the system apart.

We begin by proving that ROTC-TRANSFORMABILITY can be decided in deterministic polynomial space.

Theorem 4.17. ROTC-TRANSFORMABILITY $\in$ **PSPACE**.

Proof. We first present a *nondeterministic* Turing machine (NTM) N that decides ROTC-TRANSFORMABILITY in polynomial space. N takes as input two shapes A and B, both consisting of n nodes and at most $4n$ edges. A reasonable representation is in the form of a binary $n \times n$ matrix (representing a large enough sub-area of the grid) where an entry is 1 iff the corresponding position is occupied by a node. Given the present configuration C, where $C = A$ initially, N nondeterministically picks a valid rotation movement of a single node. This gives a new configuration C'. Then N replaces the previous configuration with C' in its memory, by setting $C \leftarrow C'$. Moreover, N maintains a counter *moves* (counting the number of moves performed so far), with maximum value equal to the total number of possible shape configurations, which is at most 2^{n^2} in the binary matrix encoding of configurations. To set up such a counter,

N just have to reserve for it n^2 (binary) tape-cells, all initialized to 0. Every time N makes a move, as above, after setting a value to C' it also increases *moves* by 1, i.e., sets $moves \leftarrow moves+1$. Then N takes another move and repeats. If it ever holds that $C' = B$ (may require N to perform a polynomial-space pattern matching on the $n \times n$ matrix to find out), then N accepts. If it ever holds that the counter is exhausted, that is, all its bits are set to 1, N rejects. If A can be transformed to B, then there must be a transformation beginning from A and producing B, by a sequence of valid rotations, without ever repeating a shape. Thus, some branch of N's computation will follow such a sequence and accept, while all non-accepting branches will reject after at most 2^{n^2} moves (when *moves* reaches its maximum value). If A cannot be transformed to B, then all branches will reject after at most 2^{n^2} moves. Thus, N correctly decides RotC-Transformability. Every branch of N, at any time, stores at most two shapes (the previous and the current), which requires $O(n^2)$ space in the matrix representation, and a 2^{n^2}-counter which requires $O(n^2)$ bits. It follows that every branch uses space polynomial in the size of the input. So, far we have proved that RotC-Transformability is decidable in nondeterministic polynomial (actually, linear) space. By applying Savitch's theorem [117] [7], we conclude that RotC-Transformability is also decidable in deterministic polynomial space (actually, quadratic), i.e., it is in **PSPACE**. □

Recall that in the line folding problem, the initial shape is a (connected) horizontal line of any even length n, with nodes $u_1, u_2, \ldots, u_n$, and the transformation asks to fold the line onto itself, forming a double-line of length $n/2$ and width 2. As part of the proof of Theorem 4.12, it was shown that if $n > 4$, then it is impossible to solve the problem by rotation only (if $n = 4$, it is trivially solved, just by rotating each endpoint above its unique neighbor). In the next proposition, we employ again the idea of a seed to show that with a little external help the transformation becomes feasible.

Proposition 4.18. *If there is a 3-line seed v_1, v_2, v_3, horizontally aligned over nodes u_3, u_4, u_5 of the line, then the line can be folded without breaking connectivity.*

Proof. We distinguish two cases, depending on whether we want the seed to be part of the final folded line or not. If yes, then we can either use a 4-line seed directly, over nodes u_3, u_4, u_5, u_6, or a 3-line seed but require n to be odd (so that $n + 3$ is even). If not, then n must be even. We show the transformation for the first case, with n odd and a 3-line seed (the other cases can be then treated with minor modifications).

We first show a simple reduction from an odd line with a 3-line seed starting over its third node to an even line with a 4-line seed starting over its third node. By rotating u_1 clockwise over u_2, we obtain the 4-line seed u_1, v_1, v_2, v_3. It only remains to move the

[7] Informally, Savitch's theorem establishes that any NTM that uses $f(n)$ space can be converted to a deterministic TM that uses only $f^2(n)$ space. Formally, it establishes that for any function $f\colon \mathbb{N} \to \mathbb{N}$, where $f(n) \geq \log n$, $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(f^2(n))$.

whole seed two positions to the right (by rotating each of its 2-lines clockwise around themselves). In this manner, we obtain an even-length line $u_2, \ldots, u_n$ and a 4-line seed starting over its third node, without breaking connectivity. Therefore, in what follows we may assume w.l.o.g. (due to the described reduction) that the initial shape is an even-length line $u_1, u_2, \ldots, u_n$ with a 4-line seed v_1, v_2, v_3, v_4 horizontally aligned over nodes u_3, u_4, u_5, u_6. Now, Figure 4.10 gives a step-by-step procedure (call it a *phase*) to exploit this 4-line seed in order to remove two nodes from the leftmost part of the line (e.g., nodes u_1 and u_2 in Figure 4.10) and add them to the line to be formed over the original line (i.e., initially two nodes shall be transferred over the rightmost part of the line; nodes v_1 and v_4 in Figure 4.10). By the end of a phase, the left part of the shape is identical in structure to the original one, having the 4-line seed at exactly the same position relative to the line as before, therefore the same procedure can be repeated over and over until the line has been completely folded. □

As already shown in Theorem 4.12, the connectivity-preservation constraint increases the class of infeasible transformations. As highlighted in Proposition 4.18, a convenient turnaround in such cases could be to introduce a suitable seed that can assist the transformation. So, for instance, in Proposition 4.18 we circumvented the impossibility of folding a line $u_1, u_2, \ldots, u_n$ in half, by adding a 3-line seed v_1, v_2, v_3, horizontally aligned over nodes u_3, u_4, u_5 of the line. Interestingly, adding the seed over nodes u_4, u_5, u_6 does not work. Therefore, given an infeasible transformation, a natural next question is to find a minimum seed (could be any connected small shape, not necessarily a line) and a placement of that seed that enables the transformation (*Minimum-Seed-Determination* problem). In the following theorem we try to identify a minimum seed that can walk the perimeter of any shape. We leave as an interesting open problem whether such a shape is able to move nodes gradually to a predetermined position, in order to transform the initial shape into a line-with-leaves (as in Theorem 4.13, but without ever breaking connectivity this time). [8]

Theorem 4.19. *If connectivity must be preserved: (i) Any* (≤ 4)*-seed cannot traverse the perimeter of a line, (ii) A 6-seed can traverse the perimeter of any orthogonally convex shape.*

4.5 Rotation and Sliding

In this section, we study the combined effect of rotation and sliding movements. We begin by proving (in Theorem 4.23) that rotation and sliding together are *transformation-universal*, meaning that they can transform any given shape to any other shape of the

[8] Another way to view this, is as an attempt to simulate the universal transformations based on combined rotation and sliding (presented in Section 4.5), in which single nodes are able to walk the perimeter of the shape.

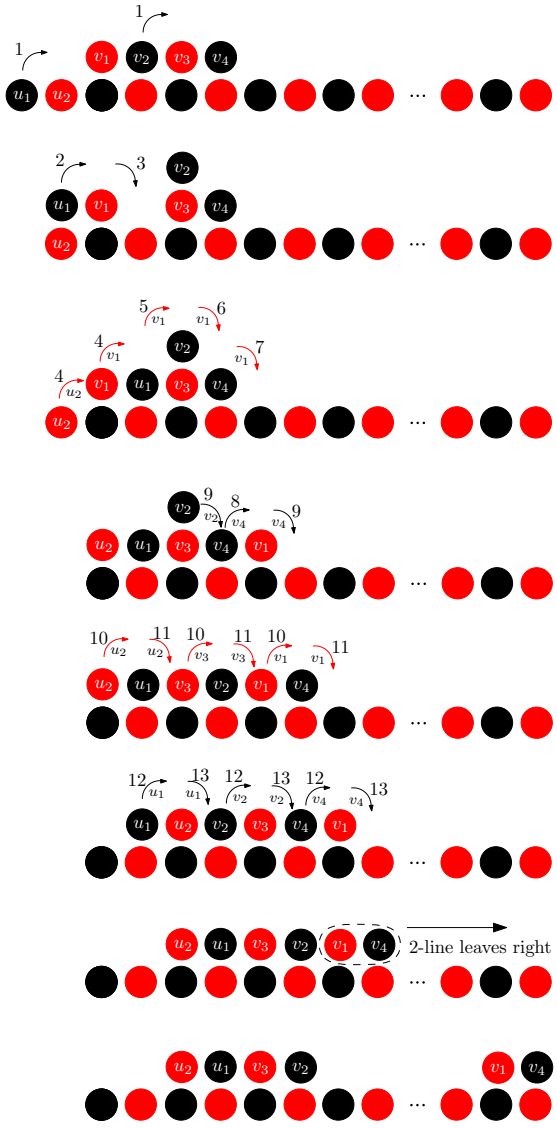

FIGURE 4.10: The main subroutine (i.e., a phase) of line folding with connectivity preservation.

same size without ever breaking the connectivity during the transformation. It would be useful for the reader to recall Definitions 4.2, 4.3, 4.4, and 4.5 and Proposition 4.6, from Section 4.2, as the results that follow make extensive use of them.

As the perimeter is a (connected) polygon, it can be traversed by a "particle" walking on its edges (the unit-length segments). We call it a particle as an imaginary entity that can walk through the edges of the grid and distinguish from nodes of the shape that may move form cell to cell. We now show how to "simulate" the particle's movement and traverse the cell-perimeter by a node, using rotation and sliding only. If needed, the reader is encouraged to refresh notions like "line segment" and "perimeter" by looking at Figure 4.3 in Section 4.2. Additionally, staying consistent with Figure

4.3, we partition the cells of the grid into red, which are the cells of the cell-perimeter, and black, which are the cells occupied by the shape.

Lemma 4.20. *If we place a node u on any position of the cell-perimeter of a connected shape A, then u can walk the whole cell-perimeter and return to its original position by using only rotations and slidings.*

Proof. We show how to "simulate" the walk of a particle moving on the edges of the perimeter. The simulation implements the following simple rules:

1. If both the current line segment traversed by the particle and the line segment traversed in the previous step correspond to edges of the same red cell, then stay put.
2. If not:
   (a) If the two consecutive line segments traversed form a line segment of length 2, then move by sliding one position in the same direction as the particle.
   (b) If the two consecutive line segments traversed are perpendicular to each other, then move by a single rotation in the same direction as the particle. [9]

It remains to prove that u can indeed always perform the claimed movements. (1) is trivial. The node temporarily stops at a "corner" and always moves in the next step according to condition 2. For (2.a), a line segment of length 2 on the perimeter is always defined by two consecutive blacks to the interior and two consecutive empty cells to the exterior (belonging to the cell-perimeter), therefore, u can slide on the empty cells. For (2.b), there must be a black in the internal angle defined by the line segments (because, by Definition 4.2 we have colored black all cells in the interior of the perimeter, that is the shape itself plus its holes if any) and an empty cell diagonally to it, in the exterior (for an example, see the right black node on the highest row containing nodes of A, in Figure 4.3, Section 4.2). Therefore, rotation can be performed. □

Next, we shall prove that u need not be an additional node, but actually a node belonging to the shape, and in particular one of those lying on the shape's boundary.

Lemma 4.21. *Let A be a connected shape of order at least 2. Then there is a subset R of the nodes on A's external surface, such that $|R| \geq 2$ and for all $u \in R$, if we completely remove u from A, then the resulting shape $A' = A - \{u\}$ is also connected.*

Proof. If the extended external surface of A contains a cycle, then such a cycle must necessarily have length at least 4 (due to geometry). In this case, any node of the

[9] Observe that when these perpendicular line segments correspond to edges of the same red cell, then the node will not rotate but will instead stay put due to prior execution of rule 1.

intersection of the external surface (non-extended) and the cycle can be removed without breaking A's connectivity. If the extended external surface of A does not contain a cycle, then it corresponds to a tree graph which by definition has at least 2 leaves, i.e., nodes of degree exactly 1. Any such leaf can be removed without breaking A's connectivity. In both cases, $|R| \geq 2$. □

Lemma 4.22. *Pick any $u \in R$ (R defined on a connected shape A as above). Then u can walk the whole cell-perimeter of $A' = A - \{u\}$ by rotations and slidings.*

Proof. It suffices to observe that u already lies on the cell-perimeter of A'. Then, by Lemma 4.20, it follows that such a walk is possible. □

We are now ready to state and prove the universality theorem of rotations and slidings. As already mentioned, this result was first proved in [43], but we here give our independently developed proof.

Theorem 4.23 ([43])**.** *Let A and B be any connected shapes, such that $|A| = |B| = n$. Then A and B can be transformed to each other by rotations and slidings, without breaking the connectivity during the transformation.*

Proof. It suffices to show that any connected shape A can be transformed to a spanning line L using only rotations and slidings and without breaking connectivity during the transformation. If we show this, then A can be transformed to L and B can be transformed to L (as A and B have the same order, therefore correspond to the same spanning line L), and by reversibility of these movements, A and B can be transformed to each other via L.

Pick the rightmost column of the grid containing at least one node of A, and consider the lowest node of A in that column. Call that node u. Observe that all cells to the right of u are empty. Let the cell of u be (i, j). The final constructed line will start at (i, j) and end at $(i, j + n - 1)$.

The transformation is partitioned into $n - 1$ phases. In each phase k, we pick a node from the original shape and move it to position $(i, j + k)$, that is, to the right of the right endpoint of the line formed so far. In phase 1, position $(i, j + 1)$ is a cell of the cell-perimeter of A. So, even if it happens that u is a node of degree 1, by Lemma 4.21, there must be another such node $v \in A$ that can walk the whole cell-perimeter of $A' = A - \{v\}$ (the latter, due to Lemma 4.22). As $u \neq v$, $(i, j + 1)$ is also part of the cell-perimeter of A', therefore, v can move to $(i, j+1)$ by rotations and slidings. As A' is connected (by Lemma 4.21), $A' \cup \{(i, j+1)\}$ is also connected and also all intermediate shapes were connected, because v moved on the cell-perimeter and, therefore, it never disconnected from the rest of the shape during its movement.

In general, the transformation preserves the following invariant. At the beginning of phase k, $1 \leq k \leq n-1$, there is a connected shape $S(k)$ (where $S(1) = A$) to the left

of column j (j inclusive) and a line of length $k-1$ starting from position $(i, j+1)$ and growing to the right. Restricting attention to $S(k)$, there is always a $v \neq u$ that could (hypothetically) move to position $(i, j+1)$ if it were not occupied. This implies that before the final movement that would place v on $(i, j+1)$, v must have been in $(i+1, j)$ or $(i+1, j+1)$, if we assume that v always walks in the clockwise direction. Observe now that from each of these positions v can perform zero or more right slidings above the line in order to reach the position above the right endpoint $(i, j+k-1)$ of the line. When this occurs, a final clockwise rotation makes v the new right endpoint of the line. The only exception is when v is on $(i+1, j+1)$ and there is no line to the right of (i, j) (this implies the existence of a node on $(i+1, j)$, otherwise connectivity of $S(k)$ would have been violated). In this case, v just performs a single downward sliding to become the right endpoint of the line. □

Theorem 4.24. *The transformation of Theorem 4.23 requires $\Theta(n^2)$ movements in the worst case.*

Proof. A *staircase* is defined as a shape of the form $(i, j), (i-1, j), (i-1, j+1), (i-2, j+1), (i-2, j+2), (i-3, j+2), \ldots$. Consider such a staircase shape of order n, as depicted in Figure 4.11. The strategy of Theorem 4.23 will choose to construct the line to the right of node u. The only node that can be selected to move in each phase without breaking the shape's connectivity is the top-left node. Initially, this is v, which must perform $\lceil n/2 \rceil$ movements to reach its position to the right of u. In general, the total number of movements M, performed by the transformation of Theorem 4.23 on the staircase, is given by

$$\begin{aligned} M &= \left\lceil \frac{n}{2} \right\rceil + 2 \cdot \sum_{i=1}^{(n-3)/2} \left(\left\lceil \frac{n}{2} \right\rceil + i \right) \\ &= \left\lceil \frac{n}{2} \right\rceil (n-2) + 2 \cdot \sum_{i=1}^{(n-3)/2} i \\ &= \Theta(n^2). \end{aligned}$$

□

Theorem 4.24 shows that the above generic strategy is slow in some cases, as is the case of transforming a staircase shape into a spanning line. We shall now show that there are pairs of shapes for which any strategy and not only this particular one, may require a quadratic number of steps to transform one shape to the other.

Definition 4.25. Define the *potential of a shape* A as its minimum "distance" from the line L, where $|A| = |L|$. The *distance* is defined as follows: Consider any placement of L relative to A and any pairing of the nodes of A to the nodes of the line. Then

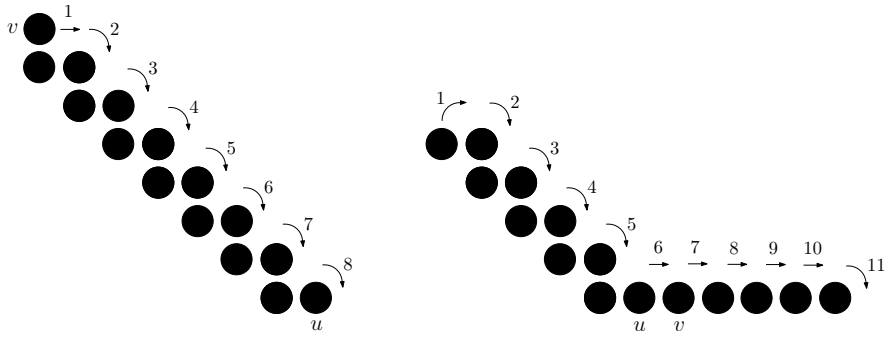

FIGURE 4.11: Transforming a staircase into a spanning line.

sum up the Manhattan distances [10] between the nodes of each pair. The minimum sum between all possible relative placements and all possible pairings is the distance between A and L and also A's potential. [11] In case the two shapes do not have an equal number of nodes, then any matching is not perfect and the distance can be defined as infinite.

Observe that the potential of the line is 0 as it can be totally aligned on itself and the sum of the distances is 0.

Lemma 4.26. *The potential of the staircase is* $\Theta(n^2)$.

Proof. We prove it for horizontal placement of the line, as the vertical case is symmetric. Any such placement leaves either above or below it at least half of the nodes of the staircase (maybe minus 1). W.l.o.g. let it be above it. Every two nodes, the height increases by 1, therefore there are 2 nodes at distance 1, 2 at distance 2,. . ., 2 at distance n/4. Any matching between these nodes and the nodes of the line gives for every pair a distance at least as large as the vertical distance between the staircase's node and the line, thus, the total distance is at least $2 \cdot 1 + 2 \cdot 2 + ... + 2 \cdot (n/4) = 2 \cdot (1 + 2 + ... + n/4) = (n/4) \cdot (n/4 + 1) = \Theta(n^2)$. We conclude that the potential of the staircase is $\Theta(n^2)$. □

Theorem 4.27. *Any transformation strategy based on rotations and slidings which performs a single movement per step requires* $\Theta(n^2)$ *steps to transform a staircase into a line.*

Proof. To show that $\Omega(n^2)$ movements are needed to transform the staircase into a line, it suffices to observe that the difference in their potentials is that much and that one rotation or one sliding can decrease the potential by at most 1. □

[10] The Manhattan distance between two points (i, j) and (i', j') is given by $|i - i'| + |j - j'|$.

[11] To make this constructive, it is sufficient to restrict the possible placements of the line to those were at least one node of the line overlaps with the shape. Note that even without this, the potential is finite and well defined as it corresponds to a placement that minimizes the above nonnegative sum.

Remark 4.28. The above lower bound is independent of connectivity preservation. It is just a matter of the total distance based on single distance-one movements.

Finally, it is interesting to observe that such lower bounds can be computed in polynomial time, because there is a polynomial-time algorithm for computing the distance between two shapes.

Proposition 4.29. *Let A and B be connected shapes. Then their distance $d(A, B)$ can be computed in polynomial time.*

Proof. The algorithm picks a node $u \in B$, a cell c of the grid occupied by a node $v \in A$, and an orientation $o \in north, east, south, west$ and draws a copy of the shape B, starting with u on c and respecting the orientation o. Then, it constructs (in its memory) a complete weighted bipartite graph (X, Y), where X and Y are equal to the node-sets of A and B, respectively. The weight $w(x, y)$ for $x \in X$ and $y \in Y$ is defined as the distance from x to y (given the drawing of shape B relative to shape A). To compute the minimum total distance pairing of the nodes of A and B for this particular placement of A and B, the algorithm computes a minimum cost perfect matching of (X, Y), e.g., by the Kuhn-Munkres algorithm (a.k.a. the Hungarian algorithm) [118], and the sum k of the weights of its edges, and sets $dist = \min\{dist, k\}$, initially $dist = \infty$. Then the algorithm repeats for the next selection of $u \in B$, cell c occupied by a node $v \in A$, and orientation o. In the end, the algorithm gives $dist$ as output. To see that $dist = d(A, B)$, observe that the algorithm just implements the procedure for computing the distance, of Definition 4.25, with the only differences being that it does not check all pairings of the nodes, instead directly computes the minimum-cost pairing, and that it does not try all relative placements of A and B but only those in which A and B share at least one cell of the grid. To see that this selection is w.l.o.g., assume that a placement of A and B in which no cell is shared achieves the minimum distance and observe that, in this case, A could be shifted one step "closer" to B, strictly decreasing their distance and, thus, contradicting the optimality of such a placement. As the different possible relative placements of A and B are $4n^2$ (place each node of B on every node of A and for each such placement there are 4 possible orientations) and the Kuhn-Munkres algorithm is a polynomial-time algorithm (in the size of the bipartite graph), we conclude that the algorithm computes the distance in polynomial time. □

To give a faster transformation either pipelining must be used (allowing for more than one movement in parallel) or more complex mechanisms that move sub-shapes consisting of many nodes, in a single step. In what remains, we follow the former approach by allowing an unbounded number of rotation and/or sliding movements to occur simultaneously in a single step (though, in pairwise disjoint areas).

4.5.1 Parallelizing the Transformations

We now maintain the connectivity preservation requirement but allow an unbounded number of rotation and/or sliding movements to occur simultaneously in a single step.

Proposition 4.30. *There is a pipelining strategy that transforms a staircase into a line in $O(n)$ parallel time.*

Proof. Number the nodes of the staircase 1 through n starting from the top and following the staircase's connectivity until the bottom-right node is reached. This gives an odd-numbered upper diagonal and an even-numbered lower diagonal. Node 1 moves as in Theorem 4.23. Any even node w starts moving as long as its upper odd neighbor has reached the same level as w (e.g., node 2 first moves after node 1 has arrived to the right of node 3). Any odd node $z > 1$ starts moving as long as its even left neighbor has moved one level down (e.g., node 3 first moves after node 2 has arrived to the right of 5). After a node starts moving, it moves in every step as in Theorem 4.23 (but now many nodes can move in parallel, implementing a pipelining strategy). It can be immediately observed that any node i starts after at most 3 movements of node $i-1$ (actually, only 2 movements for even i), so after roughly at most $3n$ steps, node $n-2$ starts. Moreover, a node that starts, arrives at the right endpoint of the line after at most n steps, which means that after at most $4n = O(n)$ steps all nodes have taken their final position in the line. □

Proposition 4.30 gives a hint that pipelining could be a general strategy to speed-up transformations. We next show how to generalize this technique to any possible pair of shapes.

Theorem 4.31. *Let A and B be any connected shapes, such that $|A| = |B| = n$. Then there is a pipelining strategy that can transform A to B (and inversely) by rotations and slidings, without breaking the connectivity during the transformation, in $O(n)$ parallel time.*

Proof. The transformation is a pipelined version of the sequential transformation of Theorem 4.23. Now, instead of picking an arbitrary next candidate node of $S(k)$ to walk the cell-perimeter of $S(k)$ clockwise, we always pick the rightmost clockwise node $v_k \in S(k)$, that is, the node that has to walk the shortest clockwise distance to arrive at the line being formed. This implies that the subsequent candidate node v_{k+1} to walk, is always "behind" v_k in the clockwise direction and is either already free to move or is enabled after v_k's departure. Observe that after at most 3 clockwise movements, v_k can no longer be blocking v_{k+1} on the (possibly updated) cell-perimeter. Moreover, the clockwise move of v_{k+1} only introduces a gap in its original position, therefore it only affects the structure of the cell-perimeter "behind" it. The strategy is to start the walk

of node v_{k+1} as soon as v_k is no longer blocking its way. As in Proposition 4.30, once a node starts, it moves in every step, and again any node arrives at the end of the line being formed after at most n movements. It follows, that if the pipelined movement of nodes cannot be blocked in any way, after $4n = O(n)$ steps all nodes must have arrived at their final positions. Observe now that the only case in which pipelining could be blocked is when a node is sliding through a (necessarily dead-end) "tunnel" of height 1 (such an example is the red tunnel on the third row from the bottom, in Figure 4.3 of Section 4.2). To avoid this, the nodes shortcut the tunnel, by visiting only its first position (i, j) and then simply skipping the whole walk inside it (that walk would just return them to position (i, j) after a number of steps). □

We next show that even if A and B are labeled shapes, that is, their nodes are assigned the indices $1, \ldots, n$ (uniquely, i.e., without repetitions), we can still transform the labeled A to the labeled B with only a linear increase in parallel time. We only consider transformations in which the nodes never change indices in any way (e.g., cannot transfer them, or swap them), so that each particular node of A must eventually occupy (physically) a particular position of B (the one corresponding to its index).

Corollary 4.32. *The labeled version of the transformation of Theorem 4.31 can be performed in $O(n)$ parallel time.*

Proof. Recall from Theorem 4.23 that the line was constructed to the right of some node u. That node was the lowest node in that column, therefore, there is no node below u in that column. The procedure of Theorem 4.31, if applied on the labeled versions of A and B will result in two (possibly differently) labeled lines, corresponding to two permutations of $1, 2, \ldots, n$, call them π_A and π_B. It suffices to show a way to transform π_A to π_B in linear parallel time, as then labeled A is transformed to π_A, then π_A to π_B, and then π_B to B (by reversing the transformation from B to π_B), all in linear parallel time.

To do this, we actually slightly modify the procedure of Theorem 4.31, so that it does not construct π_A in the form of a line, but in a different form that will allow us to quickly transform it to π_B without breaking connectivity. What we will construct is a double line, with the upper part growing to the right of node u as before and the lower part starting from the position just below u and also growing to the right. The upper line is an unordered version of the left half of π_B and the lower line is an unordered version of the right half of π_B. To implement the modification, when a node arrives above u, as before, if it belongs to the upper line, it goes to the right endpoint of the line as before, while if it belongs to the lower line, it continues its walk in order to reach the right endpoint of the lower line.

When the transformation of labeled A to the folded line is over, the procedure has to order the nodes of the folded line and then unfold in order to produce π_B. We first

order the upper line in ascending order. While we do this, the lower line stays still in order to preserve the connectivity. When we are done, we order the lower line in descending order, now keeping the upper line still. Finally, we perform a parallel right sliding of the lower line (requiring linear parallel time), so that its inverse permutation ends up to the right of the upper line, thus forming π.

It remains to show how the ordering of the upper line can be done in linear parallel time without breaking connectivity. To do this, we simulate a version of the odd-even sort algorithm (a.k.a. parallel bubble sort) which sorts a list of n numbers with $O(n)$ processors in $O(n)$ parallel time. The algorithm progresses in odd and even phases. In the odd phases, the odd positions are compared to their right neighbor and in the even phases to their left neighbor and if two neighbors are ever found not to respect the ordering a swap of their values is performed. In our simulation, we break each phase into two subphases as follows. Instead of performing all comparisons at once, as we cannot do this and preserve connectivity, in the first subphase we do every second of them and in the second subphase the rest so that between any pair of nodes being compared there are 2 nodes that are not being compared at the same time. Now if the comparison between the i-th and the $(i+1)$-th node indicates a swap, then $i+1$ rotates over $i+2$, i slides right to occupy the previous position of $i+1$, and finally $i+1$ slides left over i and then rotates left around i to occupy i's previous position. This swapping need 4 steps and does not break connectivity. The upper part has $n/2$ nodes, each subphase takes 4 steps to swap everyone (in parallel), each phase has 2 sub-phases, and $O(n)$ phases are required for the ordering to complete, therefore, the total parallel time is $O(n)$ for the upper part and similarly $O(n)$ for the lower part. This completes the proof. □

An immediate observation is that a linear-time transformation does not seem satisfactory for all pairs of shapes. To this end, take a square S and rotate its top-left corner u one position clockwise, to obtain an almost-square S'. Even though, a single counter-clockwise rotation of u suffices to transform S' to S, the transformation of Theorem 4.31 may go all the way around and first transform S' into a line and then transform the line to S. In this particular example, the distance between S and S', according to Definition 4.25, is 2, while the generic transformation requires $\Theta(n)$ parallel time. So, it is plausible to ask if any transformation between two shapes A and B can be performed in time that grows as a function of their distance $d(A, B)$. We show that this cannot always be the case, by presenting two shapes A and B with $d(A, B) = 2$, such that A and B require $\Omega(n)$ parallel time to be transformed to each other.

Proposition 4.33. *There are two shapes A and B with $d(A, B) = 2$, such that A and B require $\Omega(n)$ parallel time to be transformed to each other.*

Proof. The two shapes, a black and a red one, are depicted in Figure 4.12. Both shapes form a square which is empty inside and also open close to the middle of its bottom side. The difference between the two shapes is the positioning of the bottom "door" of length 2. The red shape has it exactly in the middle of the side, while the black shape has it shifted one position to the left. Equivalently, the bottom side of the red shape is "balanced", meaning that it has an equal number of nodes in each side of the vertical dashed axis that passes through the middle of the bottom, while the black shape is "unbalanced" having one more node to the right of the vertical axis than to its left.

To transform the black shape into the red one, a black node must necessary cross either the vertical or the horizontal axis (e.g., the black node u to move all the way around and end up at the same cell as the red node v). Because, if nothing of the two happens, then, no matter the transformation, we won't be able to place the axes so that the running shape has two pairs of balanced quadrants, while, on the other hand, the red shape satisfies this, by pairing together the two bottom quadrants and the two upper quadrants. Clearly, no move can be performed in the upper quadrants initially, as this would break the shape's connectivity. The only black nodes that can move initially are u and w and no other node can ever move unless first approached by some other node that could already move. Observe also that u and w cannot cross the vertical boundary of their quadrants, unless with help of other nodes. But the only way for a second node to move in any of these quadrants (without breaking connectivity) is for either u or w to reach the corner of their quadrant which takes at least $n/8-2$ steps and then another $n/8$ steps for any (or both) of these nodes to reach the boundary, that is, at least $n/4-2$ steps, which already proves the required $\Omega(n)$ parallel-time lower bound (even a parallel algorithm has to pay the initial sequential movement of either u or w). □

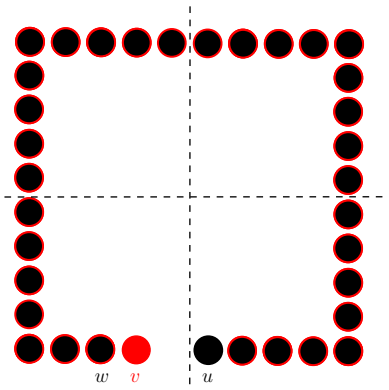

FIGURE 4.12: Counterexample for distance.

4.6 Conclusion

In this Chapter, we considered a programmable matter model where each node of the network occupies a cell of a 2D grid and the nodes are able to perform valid movements in order to transform into different shapes. We intentionally restricted attention to very minimal actuation mechanisms, namely rotation and sliding. When only rotation is available, we showed that deciding whether two given shapes can transform to each other, is in P but adding the extra constraint of maintaining connectivity makes the problems much more difficult. Adding the sliding movement allows any shape A to transformed into any shape B (provided that they have the same size). More sophisticated mechanical operations would enable a larger set of transformations and possibly also reduce the time complexity. Such an example is the ability of a node to become inserted between two neighboring nodes (while pushing them towards opposite directions). This could enable parallel mergings of two lines of length $n/2$ into a line of length n in a single step (an, thus, for example, transforming a square to a line in polylogarithmic time). Another, is the capability of rotating whole lines of nodes (like rotating arms, see, e.g., [14]). The geometry of the individual modules seems to be a parameter that greatly affects the transformation capabilities of the system as well as the algorithmic solutions. It would be interesting to extend the existing studies and problems to 3-dimensional settings (even motivated by real systems such as the Catoms3D system [119]).

In the transformations considered, there was no *a priori* constraint on the maximum area that a transformation is allowed to cover or on the maximum dimensions that its intermediate shapes are allowed to have. It seems in general harder to achieve a particular transformation if any of these restrictions is imposed. For example, the generic transformation requires some additional space below the shape and the subsequent transformations transform any shape first to a spanning line, whose maximum dimension is n, even though the original shape could have a maximum dimension as small as $\sqrt{n}$. Another interesting fact about restricting the boundaries is that in this way we get models equivalent to several interesting puzzles. For example, if the nodes are labeled, the initial shape is a square with a single empty cell, and the boundaries are restricted to the dimensions of the square, we get a generalization of the famous 15-puzzle (see, e.g., [111] for a very nice exposition of this and many more puzzles and 2-player games). Techniques developed in the context of puzzles could prove valuable for analyzing and characterizing discrete programmable matter systems. We also provided a distributed transformation for transforming any discrete convex shape into a line. There are various interesting variations of the model considered here, that would make sense. For example, to assume nodes that are oblivious w.r.t. their orientation or

to consider alternative communication principles, such as visibility and asynchronous communication.

There are also some promising specific technical questions. We do not yet know what is the complexity of RotC-Transformability. The fact that a 6-seed is capable of transferring pairs of nodes to desired positions suggests that shapes having such a seed in their exterior or being capable of self-extracting such a seed will possibly be able to transform to each other. Even if this turns out to be true, it is totally unclear whether transformations involving at least one of the rest of the shapes are feasible.

Chapter 5

Conclusions

In this thesis, we considered three different models for actively dynamic networks. For each model, we considered reconfiguration problems that may arise from their real-world counterpart that we are trying to investigate. In Chapter 2, we investigated a new distributed model involving a network comprised of computing entities that can activate new connections. We defined natural cost measures associated with the edge complexity of actively dynamic algorithms. We provided algorithms that optimize different time and edge complexity measures and we accompanied our algorithms with lower bounds for both the centralized and the distributed case of the problem. In Chapter 3, we considered a growing graphs model where a graph can grown by starting from a single node. We proposed algorithms for general graph classes that try to balance speed and efficiency. If someone wants super efficient growth schedules (zero excess edges), it is impossible to even find a $n^{\frac{1}{3}-\varepsilon}$-approximation of the length of such a schedule, unless P = NP. In Chapter 4, we considered a programmable matter model where each node of the network occupies a cell of a 2D grid and the nodes are able to perform valid movements in order to transform into different shapes. We showed that maintaining connectivity is a much more difficult problem and provided feasibility results for each case.

The results of this thesis can also be used to inspire additional work. The programmable model from chapter 4 has been extended and studied in subsequent work. In this paper [109], the authors consider the same model but they also introduce a "powerful" operation which allows the nodes to be pushed together on the line. Given this additional operation. the authors provide faster and more efficient algorithms for the reconfiguration problem while maintaining connectivity, along with some lower bounds. In another paper [110], the authors study the aforementioned problem in a distributed setting where the propose algorithms that transform any initial shape into a spanning line. Another paper [120], partially answers one of our open problems, where they solve the ROTC problem by showing that a 6-seed introduced at the perimeter of

any orthogonal convex shape A is sufficient to transform shape A into an orthogonal convex target shape B.

In Chapter 2, we proposed a model to study ad-hoc and wireless networks that also have some physical limitations to their connections. We proposed distributed algorithms that allow us to reconfigure the network to have more desirable properties. In chapter 4, we considered a geometric model of small robots that can move around to change their shape but in a centralized setting. It would be interesting to see whether we can extend this robots on a distributed setting. It might be possible to use/extend some of the algorithms from chapter 2 or borrow some techniques since that model imposes some limitations on the degree of each node.

Another interesting avenue consists of combining the growing graphs model with the programmable matter model in order to study biological systems since both systems model different properties of biological systems. Consider that in the human body, a single cell can start replicating when it receives a signal from the brain and create a large structure of cells that then moves around in order to fulfill a task. These two processes are abstractly described by our two models. This gives rise to the following problem: Given a target shape A, starting from a single node u, can we devise an algorithm that grows u into a shape of size $|V(A)|$ and the transforms into shape A? Additionally, while we have studied both of these problems separately on a centralized setting, it would be interesting to consider them both in a distributed setting either independently or together. One such preliminary result include the paper by Almalki and Michail [121], where the consider a geometric growing graphs model but they do not allow the nodes to move. They consider different types of growth and the provide lower bounds and algorithms for the graph growth problem. We also hope that the growing graphs model introduced in this thesis can help spark interest in the community to consider actively growing graphs. Most of the models in the research community consider growing graphs models where the nodes in the system are introduced gradually into the system by an external scheduler but this is not indicative for many real-world systems where the growth is actively managed by the system itself. For example, our work can be considered in self-healing systems [26] where the growth is used to correct the system itself.

Bibliography

[1] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 358–370, 1987. doi: 10.1109/SFCS.1987.7.

[2] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 206–219, 1988. doi: 10.1109/SFCS.1988.21938.

[3] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distrib. Comput.*, 18(4):235–253, 2006.

[4] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *42nd ACM Symposium on Theory of Computing (STOC)*, pages 513–522, 2010.

[5] Dana Angluin, James Aspnes, Jiang Chen, Yinghua Wu, and Yitong Yin. Fast construction of overlay networks. In *17th ACM symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–154, 2005.

[6] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[7] David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. In *32nd ACM Symposium on Theory of Computing (STOC)*, pages 504–513, 2000.

[8] Julien Bourgeois and Seth Copen Goldstein. Distributed intelligent mems: Progresses and perspectives. In *International Conference on ICT Innovations*, pages 15–25, 2011.

[9] Benoît Piranda and J. Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, 42:1619–1633, 2018.

[10] Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Amoebot - a new model for programmable matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, page 220–222, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328210. doi: 10.1145/2612669.2612712. URL `https://doi.org/10.1145/2612669.2612712`.

[11] David Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55(12):78–88, 2012.

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

[13] Adrian Dumitrescu, Ichiro Suzuki, and Masafumi Yamashita. Formations for fast locomotion of metamorphic robotic systems. *The International Journal of Robotics Research*, 23(6):583–593, 2004.

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

[15] Yunfeng Lu, Yi Yang, Alan Sellinger, Mengcheng Lu, Jinman Huang, Hongyou Fan, Raid Haddad, Gabriel Lopez, Alan R Burns, Darryl Y Sasaki, et al. Self-assembly of mesoscopically ordered chromatic polydiacetylene/silica nanocomposites. *Nature*, 410(6831):913–917, 2001.

[16] Xuli Chen, Li Li, Xuemei Sun, Yanping Liu, Bin Luo, Changchun Wang, Yuping Bao, Hong Xu, and Huisheng Peng. Magnetochromatic polydiacetylene by incorporation of fe3o4 nanoparticles. *Angewandte Chemie International Edition*, 50(24):5486–5489, 2011.

[17] Paul WK Rothemund. Folding dna to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

[18] Kyle Gilpin, Ara Knaian, and Daniela Rus. Robot pebbles: One centimeter modules for programmable matter through self-disassembly. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2485–2492. IEEE, 2010.

[19] Ara N Knaian, Kenneth C Cheung, Maxim B Lobovsky, Asa J Oines, Peter Schmidt-Neilsen, and Neil A Gershenfeld. The milli-motein: A self-folding chain of programmable matter with a one centimeter module pitch. In *2012 IEEE/RSJ*

International Conference on Intelligent Robots and Systems, pages 1447–1453. IEEE, 2012.

[20] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distrib. Comput.*, 20(4):279–304, 2007.

[21] James Aspnes and Eric Ruppert. An introduction to population protocols. In *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer, 2009.

[22] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. Mediated population protocols. *Theoretical Computer Science*, 412(22):2434–2450, 2011.

[23] Othon Michail and Paul G. Spirakis. Simple and efficient local codes for distributed stable network construction. *Distrib. Comput.*, 29(3):207–237, 2016.

[24] Regina O'Dell and Rogert Wattenhofer. Information dissemination in highly dynamic graphs. In *Proceedings of the 2005 joint Workshop on Foundations of Mobile Computing*, pages 104–110, 2005.

[25] Othon Michail, Ioannis Chatzigiannakis, and Paul G Spirakis. Naming and counting in anonymous unknown dynamic networks. In *15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 281–295, 2013.

[26] Thomas Hayes, Jared Saia, and Amitabh Trehan. The forgiving graph: A distributed data structure for low stretch under adversarial attack. volume 25, pages 121–130, 01 2009. doi: 10.1007/s00446-012-0160-1.

[27] Kenneth A. Berman. Vulnerability of scheduled networks and a generalization of Menger's theorem. *Networks*, 28(3):125–134, 1996.

[28] George B Mertzios, Othon Michail, and Paul G Spirakis. Temporal network optimization subject to connectivity constraints. *Algorithmica*, 81(4):1416–1449, 2019.

[29] Jessica Enright, Kitty Meeks, George B. Mertzios, and Viktor Zamaraev. Deleting edges to restrict the size of an epidemic in temporal networks. *Journal of Computer and System Sciences*, 119:60–77, 2021.

[30] Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. The complexity of finding small separators in temporal graphs. *Journal of Computer and System Sciences*, 107:72–92, 2020.

[31] Argyrios Deligkas and Igor Potapov. Optimizing reachability sets in temporal graphs by delaying. *Information and Computation*, 285:104890, 2022. ISSN 0890-5401. doi: https://doi.org/10.1016/j.ic.2022.104890. URL `https://www.sciencedirect.com/science/article/pii/S0890540122000323`.

[32] Spring Berman, Sándor P. Fekete, Matthew J. Patitz, and Christian Scheideler. Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 18331). *Dagstuhl Reports*, 8(8):48–66, 2019. ISSN 2192-5283. doi: 10.4230/DagRep.8.8.48. URL `http://drops.dagstuhl.de/opus/volltexte/2019/10235`.

[33] Othon Michail, George Skretas, and Paul G Spirakis. On the transformation capability of feasible mechanisms for programmable matter. *Journal of Computer and System Sciences*, 102:18–39, 2019.

[34] Hugo A Akitaya, Esther M Arkin, Mirela Damian, Erik D Demaine, Vida Dujmovic, Robin Flatland, Matias Korman, Belen Palop, Irene Parada, André van Renssen, et al. Universal reconfiguration of facet-connected modular robots by pivots: The $o(1)$ musketeers. In *27th Annual European Symposium on Algorithms (ESA)*, 2019.

[35] Abdullah Almethen, Othon Michail, and Igor Potapov. Pushing lines helps: Efficient universal centralised transformations for programmable matter. *Theoretical Computer Science*, 830-831:43 – 59, 2020.

[36] Seth Copen Goldstein, Jason D Campbell, and Todd C Mowry. Programmable matter. *Computer*, 38(6):99–101, 2005.

[37] Michael Andrew McEvoy and Nikolaus Correll. Materials that couple sensing, actuation, computation, and communication. *Science*, 347(6228):1261689, 2015.

[38] Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable dna self-assembly. *Nature*, 567(7748):366–372, 2019.

[39] Zahra Derakhshandeh, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. Universal coating for programmable matter. *Theoretical Computer Science*, 671:56–68, 2017.

[40] Hugo A. Akitaya, Esther M. Arkin, Mirela Damian, Erik D. Demaine, Vida Dujmović, Robin Flatland, Matias Korman, Belen Palop, Irene Parada, André van Renssen, and Vera Sacristán. Universal reconfiguration of facet-connected modular robots by pivots: The $O(1)$ musketeers. *Algorithmica*, 83(5):1316–1351, 2021.

[41] Paul WK Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the 32nd annual ACM Symposium on Theory of Computing (STOC)*, pages 459–468, 2000.

[42] Adrian Dumitrescu, Ichiro Suzuki, and Masafumi Yamashita. Motion planning for metamorphic systems: Feasibility, decidability, and distributed reconfiguration. *IEEE Transactions on Robotics and Automation*, 20(3):409–418, 2004.

[43] Adrian Dumitrescu and János Pach. Pushing squares around. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 116–123. ACM, 2004.

[44] Othon Michail, George Skretas, and Paul G. Spirakis. On the transformation capability of feasible mechanisms for programmable matter. In *ICALP*, 2017.

[45] Othon Michail, George Skretas, and Paul G. Spirakis. Distributed computation and reconfiguration in actively dynamic networks. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 448–457, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375825. doi: 10.1145/3382734.3405744. URL `https://doi.org/10.1145/3382734.3405744`.

[46] George B. Mertzios, Othon Michail, George Skretas, Paul G. Spirakis, and Michail Theofilatos. The complexity of growing a graph. *CoRR*, abs/2107.14126, 2021. URL `https://arxiv.org/abs/2107.14126`.

[47] James Aspnes and Gauri Shah. Skip graphs. *ACM Trans. Algorithms*, 3(4): 37–es, nov 2007. ISSN 1549-6325. doi: 10.1145/1290672.1290674. URL `https://doi.org/10.1145/1290672.1290674`.

[48] James Aspnes and Yinghua Wu. $o(\log n)$-time overlay network construction from graphs with out-degree 1. In *11th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 286–300, 2007.

[49] Thorsten Götte, Kristian Hinnenthal, and Christian Scheideler. Faster construction of overlay networks. In *26th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 262–276, 2019.

[50] Thorsten Götte, Kristian Hinnenthal, Christian Scheideler, and Julian Werthmann. Time-optimal construction of overlay networks. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 457–468, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385480. doi: 10.1145/3465084.3467932. URL `https://doi.org/10.1145/3465084.3467932`.

[51] Seth Gilbert, Gopal Pandurangan, Peter Robinson, and Amitabh Trehan. Dconstructor: Efficient and robust network construction with polylogarithmic overhead. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, 2020.

[52] Robert Gmyr, Kristian Hinnenthal, Christian Scheideler, and Christian Sohler. Distributed monitoring of network properties: The power of hybrid networks. In *44th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 137:1–137:15, 2017.

[53] Christian Scheideler and Alexander Setzer. On the complexity of local graph transformations. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 150:1–150:14, 2019.

[54] Michelle M Chan, Zachary D Smith, Stefanie Grosswendt, Helene Kretzmer, Thomas M Norman, Britt Adamson, Marco Jost, Jeffrey J Quinn, Dian Yang, Matthew G Jones, et al. Molecular recording of mammalian embryogenesis. *Nature*, 570(7759):77–82, 2019.

[55] Hugo A. Méndez-Hernández, Maharshi Ledezma-Rodríguez, Randy N. Avilez-Montalvo, Yary L. Juárez-Gómez, Analesa Skeete, Johny Avilez-Montalvo, Clelia De-la Peña, and Víctor M. Loyola-Vargas. Signaling overview of plant somatic embryogenesis. *Frontiers in Plant Science*, 10, 2019.

[56] Andrew Howard, Maja J. Matarić, and Gaurav S Sukhatme. An incremental self-deployment algorithm for mobile sensor networks. *Autonomous Robots*, pages 113–126, 2002.

[57] Ioannis Chatzigiannakis, Athanasios Kinalis, and Sotiris Nikoletseas. Adaptive energy management for incremental deployment of heterogeneous wireless sensors. *Theory of Computing Systems*, 42:42–72, 2008.

[58] Mário Cordeiro, Rui P. Sarmento, Pavel Brazdil, and João Gama. Evolving networks and social network analysis methods and techniques. In *Social Media and Journalism*, chapter 7. 2018.

[59] Paul WK Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

[60] David Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55(12):78–88, 2012.

[61] Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science (ITCS)*, pages 353–354, 2013.

[62] Luca Becchetti, Andrea Clementi, Francesco Pasquale, Luca Trevisan, and Isabella Ziccardi. Expansion and flooding in dynamic random networks with node churn. In *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 976–986, 2021.

[63] John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Towards robust and efficient computation in dynamic peer-to-peer networks. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, page 551–569, USA, 2012. Society for Industrial and Applied Mathematics.

[64] Luca Bombelli, Joohan Lee, David Meyer, and Rafael D Sorkin. Space-time as a causal set. *Physical Review Letters*, 59(5):521, 1987.

[65] David Porter Rideout and Rafael D Sorkin. Classical sequential growth dynamics for causal sets. *Physical Review D*, 61(2):024002, 1999.

[66] Alexei Lisitsa and Igor Potapov. On the computational power of querying the history. *Fundam. Inform.*, 91:395–409, 01 2009. doi: 10.3233/FI-2009-0049.

[67] Roy S. Rubinstein and John N. Shutt. Self-modifying finite automata: An introduction. *Information Processing Letters*, 56(4):185–190, 1995. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(95)00157-8. URL `https://www.sciencedirect.com/science/article/pii/0020019095001578`.

[68] Othon Michail and Paul G Spirakis. Elements of the theory of dynamic networks. *Communications of the ACM*, 61(2):72–72, 2018.

[69] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[70] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[71] David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences*, 64 (4):820–842, 2002.

[72] Kenneth A Berman. Vulnerability of scheduled networks and a generalization of menger's theorem. *Networks: An International Journal*, 28(3):125–134, 1996.

[73] George B Mertzios, Othon Michail, and Paul G Spirakis. Temporal network optimization subject to connectivity constraints. *Algorithmica*, 81(4):1416–1449, 2019.

[74] Jessica Enright, Kitty Meeks, George B. Mertzios, and Viktor Zamaraev. Deleting edges to restrict the size of an epidemic in temporal networks. *Journal of Computer and System Sciences*, 119:60–77, 2021.

[75] Eleni C. Akrida, George B. Mertzios, Paul G. Spirakis, and Viktor Zamaraev. Temporal vertex cover with a sliding time window. *Journal of Computer and System Sciences*, 107:108–123, 2020.

[76] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms (TALG)*, 3(4):37–es, 2007.

[77] Thorsten Götte, Kristian Hinnenthal, and Christian Scheideler. Faster construction of overlay networks. In *Proceedings of the 26th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 262–276. Springer, 2019.

[78] Thorsten Götte, Kristian Hinnenthal, Christian Scheideler, and Julian Werthmann. Time-optimal construction of overlay networks. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 457–468, 2021.

[79] Othon Michail, George Skretas, and Paul G Spirakis. Distributed computation and reconfiguration in actively dynamic networks. In *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC)*, pages 448–457, 2020.

[80] Min Chih Lin, Francisco J. Soulignac, and Jayme Luiz Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *Theoretical Computimg Science*, 426:75–90, 2012.

[81] Hans-Jürgen Bandelt and Erich Prisner. Clique graphs and helly graphs. *Journal of Combinatorial Theory, Series B*, 51(1):34–45, 1991.

[82] Tim Poston. *Fuzzy Geometry*. PhD thesis, University of Warwick, 1971.

[83] Victor Chepoi. On distance-preserving and domination elimination orderings. *SIAM Journal on Discrete Mathematics*, 11(3):414—-436, 1998.

[84] Béla Bollobás. *Random graphs*. Number 73 in Cambridge studies in advanced mathematics. Cambridge University Press, 2nd edition, 2001. ISBN 0-521-80920-7.

[85] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S." Tomkins. The web as a graph: Measurements, models, and method. In *Computing and Combinatorics*, pages 1–17. Springer Berlin Heidelberg, 1999.

[86] Christian Scheideler and Alexander Setzer. On the complexity of local graph transformations. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[87] Othon Michail, George Skretas, and Paul G Spirakis. On the transformation capability of feasible mechanisms for programmable matter. *Journal of Computer and System Sciences*, 102:18–39, 2019.

[88] David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3:103–128, 2007.

[89] Howard Williams. A linear algorithm for colouring planar graphs with five colours. *The Computer Journal*, 28:78–81, 1985.

[90] Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.

[91] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7 (4):615–633, 2008.

[92] David Doty. Timing in chemical reaction networks. In *Proc. of the 25th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 772–784, 2014.

[93] Zahra Derakhshandeh, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, page 21. ACM, 2015.

[94] Joshua J Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17(1):81–96, 2018.

[95] Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal shape formation for programmable matter. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 289–299. ACM, 2016.

[96] Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Brief Announcement: Shape Formation by Programmable Particles. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, volume 91, pages 48:1–48:3, 2017. ISBN 978-3-95977-053-8.

[97] Sándor Fekete, Andréa W Richa, Kay Römer, and Christian Scheideler. Algorithmic foundations of programmable matter (Dagstuhl Seminar 16271). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016. Also in *ACM SIGACT News*, 48.2:87-94, 2017.

[98] Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Solving the robots gathering problem. In *International Colloquium on Automata, Languages, and Programming*, pages 1181–1196. Springer, 2003.

[99] Evangelos Kranakis, Danny Krizanc, and Euripides Markou. The mobile agent rendezvous problem in the ring. *Synthesis Lectures on Distributed Computing Theory*, 1(1):1–122, 2010.

[100] Masahiro Shibata, Toshiya Mega, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Uniform deployment of mobile agents in asynchronous rings. *Journal of Parallel and Distributed Computing*, 119:92–106, 2018.

[101] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed computing by oblivious mobile robots. *Synthesis Lectures on Distributed Computing Theory*, 3(2):1–185, 2012.

[102] Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. Comput.*, 28(4):1347–1363, March 1999. ISSN 0097-5397. doi: 10.1137/S009753979628292X. URL `http://dx.doi.org/10.1137/S009753979628292X`.

[103] Shantanu Das, Paola Flocchini, Nicola Santoro, and Masafumi Yamashita. Forming sequences of geometric patterns with oblivious mobile robots. *Distributed Computing*, 28(2):131–145, April 2015.

[104] Alejandro Cornejo, Fabian Kuhn, Ruy Ley-Wild, and Nancy Lynch. Keeping mobile robot swarms connected. In *Proceedings of the 23rd International*

Symposium on Distributed computing, DISC'09, pages 496–511, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 3-642-04354-2, 978-3-642-04354-3. URL http://dl.acm.org/citation.cfm?id=1813164.1813227.

[105] Zack Butler, Keith Kotay, Daniela Rus, and Kohji Tomita. Generic decentralized control for lattice-based self-reconfigurable robots. *The International Journal of Robotics Research*, 23(9):919–937, 2004.

[106] Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):43–52, 2007.

[107] Greg Aloupis, Nadia Benbernou, Mirela Damian, Erik D Demaine, Robin Flatland, John Iacono, and Stefanie Wuhrer. Efficient reconfiguration of lattice-based modular robots. *Computational geometry*, 46(8):917–928, 2013.

[108] Yukiko Yamauchi, Taichi Uehara, and Masafumi Yamashita. Brief announcement: pattern formation problem for synchronous mobile robots in the three dimensional euclidean space. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 447–449. ACM, 2016.

[109] Abdullah Almethen, Othon Michail, and Igor Potapov. On efficient connectivity-preserving transformations in a grid. *Theoretical Computer Science*, 898:132–148, 2022. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2021.11.004. URL https://www.sciencedirect.com/science/article/pii/S0304397521006514.

[110] Abdullah Almethen, Othon Michail, and Igor Potapov. Distributed transformations of hamiltonian shapes based on line moves. In *Algorithms for Sensor Systems*, pages 1–16. Springer International Publishing, 2021.

[111] Erik D Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *International Symposium on Mathematical Foundations of Computer Science*, pages 18–33. Springer, 2001.

[112] Robert A Hearn and Erik D Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, 2005.

[113] Aaron T Becker, Erik D Demaine, Sándor P Fekete, Jarrett Lonsford, and Rose Morris-Wright. Particle computation: complexity, algorithms, and logic. *Natural Computing*, pages 1–21, 2017.

[114] An Nguyen, Leonidas J Guibas, and Mark Yim. Controlled module density helps reconfiguration planning. In *Proc. of 4th International Workshop on Algorithmic Foundations of Robotics*, pages 23–36, 2000.

[115] Jennifer E Walter, Jennifer L Welch, and Nancy M Amato. Distributed reconfiguration of metamorphic robot chains. *Distributed Computing*, 17(2):171–189, 2004.

[116] Camille Jordan. *Cours d'analyse de l'École polytechnique*, volume 1. Gauthier-Villars et fils, 1893.

[117] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2): 177–192, 1970. ISSN 0022-0000. doi: DOI:10.1016/S0022-0000(70)80006-X. URL `http://www.sciencedirect.com/science/article/B6WJ0-4RFM9YP-5/2/deae6aa41cbc8971f3fb3205357e5832`.

[118] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[119] Benoit Piranda and Julien Bourgeois. Geometrical study of a quasi-spherical module for building programmable matter. In *Distributed Autonomous Robotic Systems*, pages 387–400. Springer, 2018.

[120] Matthew Connor and Othon Michail. Centralised connectivity-preserving transformations by rotation: 3 musketeers for all orthogonal convex shapes, 2022. URL `https://arxiv.org/abs/2207.03062`.

[121] Nada Almalki and Othon Michail. On geometric shape construction via growth operations, 2022. URL `https://arxiv.org/abs/2207.03275`.