The Complexity of Growing a Graph

George Mertzios[a,1], Othon Michail[b], George Skretas[b,c,*], Paul G. Spirakis[b,2], Michail Theofilatos[b]

[a]*Department of Computer Science, Durham University, Durham, UK*
[b]*Department of Computer Science, University of Liverpool, Liverpool, UK*
[c]*Hasso Plattner Institute, University of Potsdam, Potsdam, Germany*

Abstract

We study a new algorithmic process of graph growth which starts from a single initial vertex and operates in discrete time-steps, called *slots*. In every slot, the graph grows via two operations (i) vertex generation and (ii) edge activation. The process completes at the last slot where a (possibly empty) subset of the edges of the graph are removed. Removed edges are called *excess edges*. The main problem investigated in this paper is: Given a target graph G, design an algorithm that outputs a process that grows G, called a *growth schedule*. Additionally, we aim to minimize the total number of slots k and of excess edges ℓ used by the process. We provide both positive and negative results, with our main focus being either schedules with sub-linear number of slots or with no excess edges.

Keywords: Dynamic graph, temporal graph, cop-win graph, graph process, polynomial-time algorithm, lower bound, NP-complete, hardness result

1. Introduction

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 [13, 32]. In human-made systems, sensor networks are being deployed incrementally to monitor a given geographic area [22, 14], social-network groups expand by connecting with new individuals [16], DNA self-assembly automatically grows molecular shapes and patterns starting from a seed assembly [36, 17, 38], and high churn or mobility can cause substantial changes in the size and structure of computer networks [7, 4]. 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 [10, 34].

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 geometric and bounded-degree. 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 causal set theory.

Further classification comes in terms of the source and control of the network dynamics. Sometimes, the dynamics is 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, 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 graph-theoretic abstraction of network-growth processes. We do not impose any strong *a priori* constraints, like geometry, on the graph structure. We restrict our attention to *centralized* control and include weak conditions on the graph dynamics, such as "locality", according to which a newly introduced vertex u' in the neighborhood of a vertex u, can only be connected to vertices within distance $d - 1$ from u. We consider two measures of efficiency, 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, one can measure the number of times cells have divided, which 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 recent work in the theory of dynamic networks [31, 28, 12]. Research on dynamic graphs studies the algorithmic and structural properties of graphs $G_t = (V_t, E_t)$, in which V_t are sets of time-vertices and E_t are sets 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 dynamic graphs in which V_t is *static*, i.e., time-invariant [23, 8, 27, 18, 39, 1]. In overlay networks [2, 3, 20, 19, 21] and distributed network reconfiguration [30], V_t is a static set of processors that control in a decentralized way the edge dynamics. Even though we do not study distributed processes, our model also has *active*, i.e., algorithmically controlled, dynamics and a 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 network models, we reveal interesting similarities to cop-win graphs [25, 5, 33, 15]. There are other well-studied models and processes of graph growth, somewhat related to our model, such as the preferential attachment model by Barabasi and Albert [6], as well as other random graph generators [9, 24]. While initiating this study from a non-geometric centralized viewpoint, we anticipate that it can inspire work on geometric models and models in which the growth process is controlled in a distributed way. Note that

centralized upper bounds can be translated into (possibly inefficient) distributed solutions, while lower bounds readily apply to the distributed case. There are other recent 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 [37] and of some of the authors of this paper on geometric transformations for programmable matter [29].

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 each 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 each 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 as measured from u'. Finally, it removes a (possibly empty) subset of edges whose removal does not disconnect the graph, before moving on to the next slot. These edge-modification operations are capturing, at a high level, the local dynamics present in most of the applications discussed previously.

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). In this case, upon being generated, u' 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. In this work, we 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.

The rest of this paper exclusively focuses on $d = 2$. 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, the process grows K_n in $n-1$ 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 delete all edges in $E(K_n) \setminus E(G)$ at the last slot. However, this process maximizes both complexity measures that we wanted to minimize; it uses $n-1$ slots and deletes 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 a maximum degree vertex of G, and Δ denotes its degree, then it is not hard to see that this process uses $n-1-\Delta$ 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 a *growth schedule* for G of at most k slots and using at most ℓ excess edges, if one exists. All algorithms we consider are polynomial-time.[3]

For an illustration of the discussion so far, consider the graph family $F_{star} = \{G \mid G$ is a star graph 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 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 $u_0 u'$, 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 1. Thus, the process grows the target star graph G in $\delta = \log n$ 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.

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 (respectively of excess edges) then the number of excess edges (respectively slots) increases. To gain some insight into this trade-off, consider the example of a path graph G on n vertices $u_0, u_1, \ldots, u_{n-1}$, 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 edges that 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 with respect to these two measures.

1.3. Contribution

Section 2 begins by presenting the model and problem statement for edge-activation distance $d = 2$. In Section 2.2, 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.

---

[3]Note that this reference to *time* is about the running time of an algorithm computing a growth schedule. However, the length of the growth schedule is another representation of time: the time required by the respective process to grow a graph. To distinguish between the two notions of time, in our results we 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.

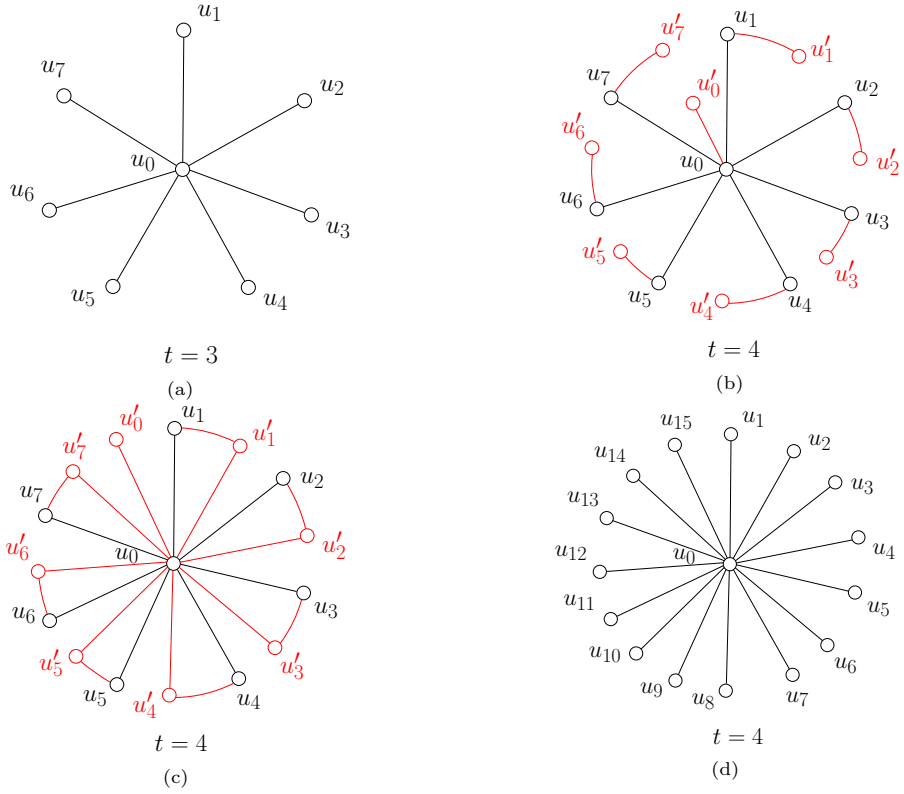Figure 1: The operations of the star graph process in slot $t = 4$. (a) A star with 2^3 vertices grown by the end of slot 3. (b) For every u_i, a vertex u'_i is generated by the process and is connected to u_i. (c) New vertices u'_i are connected to u_0. (d) Edges between peripheral-vertices are being removed to obtain the star with 2^4 vertices grown by the end of slot 4. We rename the vertices for clarity.

In Section 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 with no excess edges. We define an ordering of the vertices of a graph G, called *candidate elimination ordering* and 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 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 Section 5, we investigate cases for edge-activation distance $d = 1$ and $d \geq 3$.

In Section 6, we conclude and discuss some interesting open problems.

2. Preliminaries

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 is determined by a centralized *growth process* (or *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 the edge uu' to E_t along with any subset of the edges $\{vu' \mid v \in V_{t-1}$ is at distance at most $d - 1$ from u in $G_{t-1}\}$ (*edge-activation* operation), where $d \geq 1$ is an integer *edge-activation distance* fixed in advance. We call u' the vertex generated by the process for vertex u in slot t. We 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 that does not disconnect the graph (*edge deletion* operation). We 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 given by $V_t = V_{t-1} \cup V_t^+$ and $E_t = (E_{t-1} \cup E_t^+) \setminus E_t^-$. 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, using k slots and ℓ excess edges, where $\ell = \sum_{t=1}^{k} |E_t^-|$, i.e., ℓ is equal to the total number of deleted edges. The main problem studied in this paper is the following.

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. Throughout, the number of vertices of the target graph G is denoted by n. In this paper, computation is always to be performed by a *centralized* polynomial-time algorithm.

6

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 $B_w = (u_0, u_1, \ldots, u_{t-1}, u_t = w)$ of vertices, where $u_i \overset{i+1}{\to} u_{i+1}$, for every $i = 0, 1, \ldots, t-1$. That is, B_w is the causal order 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. We also define the sets $N(u)$ and $N[u]$ to be the neighborhood of u and closed neighborhood of u, respectively.

We now give a formal definition of a graph growth schedule. The definition is given for $d = 2$ which is the main focus of this paper. For completeness, the cases $d \neq 2$ are studied in Section 5. We will be using this description for the pseudocode of our algorithms.

Definition 1 (Growth schedule for $d = 2$). *Let $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k, \mathcal{E})$ be a sequence of sets, where $\mathcal{E}$ is a set of edges, and each $\mathcal{S}_t = \{(u_1, v_1, E_1), (u_2, v_2, E_2), \ldots, (u_q, v_q, E_q)\}$ is a set of tuples such that, for every j, where $1 \leq j \leq q$, 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 slot i, where $2 \leq t \leq k$, the following conditions are all satisfied:*

- *the sets $\{v_1, v_2, \ldots, v_q\}$ and $\{u_1, u_2, \ldots, u_q\}$ are disjoint,*

- *each vertex $v_j \in \{v_1, v_2, \ldots, v_q\}$ does not appear in any set among $\mathcal{S}_1, \ldots, \mathcal{S}_{t-1}$ (i.e., v_j is "born" at slot t),*

- *for each vertex $u_j \in \{u_1, u_2, \ldots, u_q\}$, there exists exactly one set among $\mathcal{S}_1, \ldots, \mathcal{S}_{t-1}$ which contains a tuple (u', u_j, E') (i.e., u_j was "born" at a slot before slot t).*

Let t be a slot, $2 \leq t \leq k$, and let u be a vertex that has been generated at some slot $t' \leq t$, that is, u appears in at least one tuple of a set among $\mathcal{S}_1, \ldots, \mathcal{S}_t$. We denote by E^t the union of all edge sets that appear in the tuples of the sets $\mathcal{S}_1, \ldots, \mathcal{S}_t$; E^t is the set of all edges activated until slot t. We denote by $N_t(u)$ the set of neighbors of u in E^t. If, in addition, $\mathcal{E} \subseteq E^k$ and, for every $2 \leq t \leq k$ and every $(u_j, v_j, E_j) \in \mathcal{S}_t$, we have that $N_t[v_j] \subseteq N_t[u_j]$, 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, E^k is the set of activated edges of the graph, and $\mathcal{E}$ is the set of deleted edges of the graph. We say that G has a growth schedule σ of k slots and $\ell = |\mathcal{E}|$ excess edges.[4]

2.2. Basic Properties and Sub-processes

We now give some basic properties for growing a graph G which restrict the possible growth schedules and also provide some lower bounds on the number of slots. We also provide some basic algorithms which will be used as sub-processes in the rest of the paper.

Proposition 1. *The vertices generated in a slot form an independent set in the target graph G.*

Proof. Let G_{t-1} be the graph at the beginning of slot t. Consider any pair of neighboring vertices u_1, u_2, i.e., $d(u_1, u_2) = 1$, where $d(u, v)$ denotes the distance between u and v. Assume that vertices u_1, u_2 generate vertices v_1, v_2, respectively, in slot t. The distance between vertices v_1, v_2 in slot t just after they are generated is $d(u_1, u_2) = 3$ and therefore, the process cannot activate an edge

---

[4]Note that our definition does not specify when the edges are removed from the graph. Since we do not consider the optimization criterion of minimizing excess edges that are present throughout the transformation, removing the excess edges at the end of the growth schedule is equivalent to removing them gradually throughout the schedule.

between them. The same holds true for any other pair of non-neighboring vertices, because the distance between their children is $d(u_1, u_2) > 3$. $\qquad\square$

Proposition 2. *Consider a growth schedule σ for a graph G. Let t_1, t_2, where $t_1 \leq t_2$, be the slots in which two vertices u, w are generated, respectively. Let $d(u, w)_{t_2}$ be the distance between u and w at the end of slot t_2. Then, at the end of any slot $t \geq t_2$, $d(u, w)_t \geq d(u, w)_{t_2}$.*

Proof. Given that $d = 2$, for any vertex that is generated at slot t, edges can only be activated with its parent and with the neighbors of its parent. This implies that the edge activation operations at slot t, cannot reduce the distance between two vertices u, v that have been generated in slots t_1, t_2, where $t_1, t_2 < t$, respectively. $\qquad\square$

Proposition 3. *Let t_1, t_2, where $t_1 \leq t_2$, be the slots in which two vertices u, w are generated by a growth schedule σ for a graph G, respectively, and edge uw 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 2, 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). The same holds for the children of w. All vertices that belong to the progeny P_u of u (i.e., each vertex z such that $u \in B_z$) must be at distance at least 2 from w, therefore they cannot be neighbors with any vertex in P_w. $\qquad\square$

We will now provide some lower bounds on the number of slots of any growth schedule σ for graph G. First, we also define the *chromatic number* and the *clique number* of a graph. The chromatic number of a graph G, denoted by $\chi(G)$, is the minimum number of colors needed to color the vertices of G in such a way that no two adjacent vertices receive the same color. The clique number of a graph G, denoted by $\omega(G)$, is the number of vertices in the largest clique of G.

Lemma 1. *Any growth schedule σ for a graph 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 1, 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 $\chi(G) > k$. $\qquad\square$

Lemma 2. *Any growth schedule σ for a graph G requires at least $\omega(G)$ slots.*

Proof. By Proposition 1, we know that every slot must contain an independent set of the graph and thus, it cannot contain more than one vertex from each clique. Assume that the largest clique of graph G has q vertices. By the pigeonhole principle, it follows that σ must have at least q slots. $\qquad\square$

We present simple algorithms for growing path graphs and star graphs. We use these as sub-processes in both our positive and negative results. The growth schedules returned by these algorithms use a number of slots which is logarithmic and a number of excess edges which is linear in the number of vertices 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 with respect to their number of slots. It will later follow (by Corollary 2, Section 4.3) that they are also optimal with respect to the number of excess edges used for this time-bound.
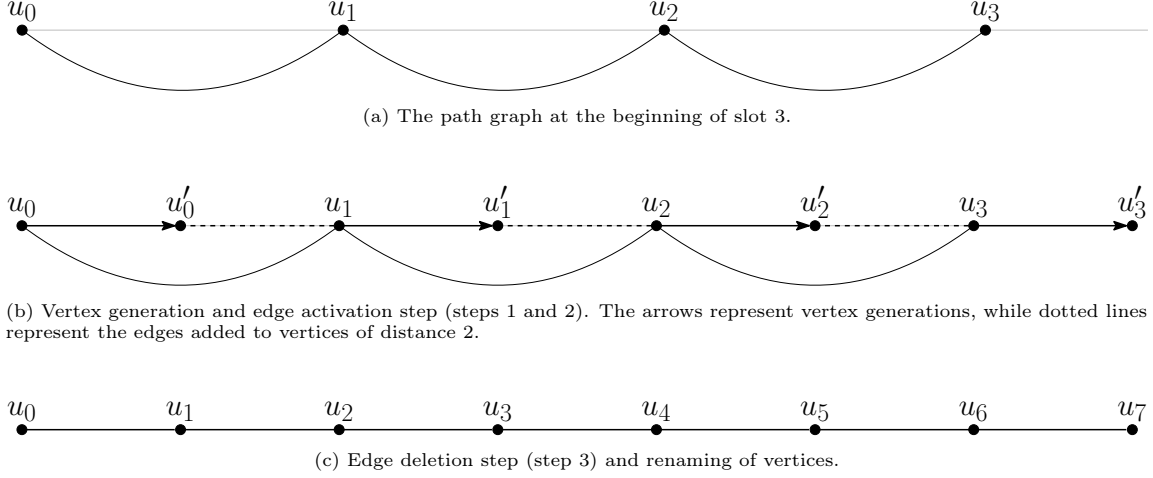
8

(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) Edge deletion step (step 3) and renaming of vertices.

Figure 2: Third slot of the $path$ algorithm.

$Path$ **algorithm:** Let u_0 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 for G as follows. For every slot $1 \leq t \leq \lceil \log n \rceil$ and every vertex $u_i \in V_{t-1}$, it generates a new vertex u_i' and activates edge $u_i'u_i$. Then, for all $0 \leq i \leq |V_{t-1}| - 2$, it activates edge $u_i'u_{i+1}$ and deletes edge u_iu_{i+1}. Finally, it renames the vertices to $u_0, u_1, \ldots, u_{2|V_{t-1}|-1}, u_{2|V_{t-1}|}$ from left to right, before moving on to the next slot. Figure 2 shows an example slot produced by the path algorithm.

Lemma 3. *For any path graph G on n vertices, the* path *algorithm computes in polynomial time a growth schedule for G of $\lceil \log n \rceil$ slots and of $n-1$ excess edges.*

Proof. In every slot, apart from the last one, for every vertex u_i the schedule returned by the algorithm generates a new vertex u_i', thus doubling the length of the path. It follows that the schedule grows a path of length n in $\lceil \log n \rceil$ slots.

For the excess edges, consider that at the end of each slot t, every edge activated in 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, which means that the edge activations are $2(n-1)$. Therefore, the excess edges are exactly $2(n-1) - (n-1) = n-1$ since the final path graph has $n-1$ edges. $\square$

$Star$ **algorithm:** The description of the algorithm can be found in Section 1.2.

Lemma 4. *For any star graph G on n vertices, the* star *algorithm computes in polynomial time a growth schedule for G of $\lceil \log n \rceil$ slots and $n - 1 - \lceil \log n \rceil$ of excess edges.*

Proof. By construction, by the end of slot t the schedule returned by the algorithm has grown a star graph of 2^t vertices. It follows that the schedule grows a star of n vertices in $\lceil \log n \rceil$ slots.

For the excess edges, in every slot and every vertex generated by a leaf, the edge between them will later be deleted. As $n - 1 - \lceil \log n \rceil$ vertices are generated by a leaf, there is a total of $n - 1 - \lceil \log n \rceil$ excess edges. $\square$

9

3. Growth Schedules of Zero Excess Edges

In this section, we study which target graphs can be grown using 0 excess edges for edge-activation distance $d = 2$. We begin by providing an algorithm that decides whether there exists a growth schedule for a graph G. We then give an algorithm that computes a schedule of $k = \log n$ slots for a target graph G, if one exists. Our main technical result shows that computing the shortest 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 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 2. *Let $G = (V, E)$ be a connected graph. A vertex $v \in V$ can be the last generated vertex in a growth schedule σ for G of no excess edges if there exists a vertex $w \in V \setminus \{v\}$ 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. The set of candidate vertices in G is denoted by S_G (see Figure 3).*

Definition 3. *A candidate elimination ordering of a connected 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 $\{v_i, v_2, \ldots, v_n\}$, for $1 \leq i \leq n$.*
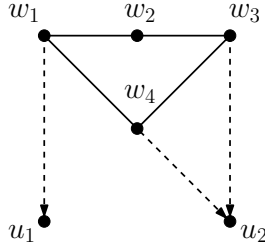

Figure 3: Let graph G_t to be the graph grown after slot t. Vertices u_1 and u_2 are candidate vertices. The arrows represent all possible vertex generations in slot t. Vertex w_1 is a candidate parent of u_1, while w_3 and w_4 are candidate parents of u_2.

Lemma 5. *A connected graph G has a growth schedule of $n - 1$ slots and no 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 there are not excess edges, this property remains true in G_{t+1}. Therefore, any vertex u generated in slot t, is a candidate vertex in graph G_{t+1}. For the reverse direction, if a graph G has a candidate elimination ordering, we can compute a growth schedule σ of $n - 1$ slots for that graph as follows: add the last vertex u in the ordering to the last slot empty slot of σ, along with the incident edges of u in G. Then remove vertex u along with its incident edges from G, and remove it from the ordering as well. Repeat the above process until graph G has a single vertex which is the initiator. The growth schedule has no excess edges since in every iteration we remove a vertex u, where $N[u] \subseteq N[w]$ for some vertex w in G. $\square$

The following algorithm decides whether a graph has a candidate elimination ordering, and therefore, whether it can be grown with a schedule of $n - 1$ slots and of no excess edges. The algorithm computes the slots of the schedule in reverse order.

10

Algorithm 1 Candidate elimination ordering algorithm.

Input: A graph $G = (V, E)$ on n vertices.
Output: A growth schedule for G, if one exists.
  1: **for** $t = n - 1$ downto 1 **do**
  2:     $\mathcal{S}_t = \emptyset$
  3:     **if** there exists a candidate vertex u **then**
  4:         $\mathcal{S}_t \leftarrow \{(u, v, \{vw : w \in N(v)\})\}$
  5:         $V \leftarrow V \setminus \{v\}$
  6:     **if** $\mathcal{S}_t = \emptyset$ **then**
  7:         **return** "no"
  8: **return** $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{n-1}, \emptyset)$

`Candidate elimination ordering` **algorithm**: Informally, given a connected graph $G = (V, E)$, in each iteration t, the algorithm finds and deletes a candidate vertex and its 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 a single vertex left at which point the algorithm outputs a growth schedule. If the algorithm cannot find any candidate vertex for removal, it outputs "no", meaning that the graph cannot be grown. See Algorithm 1 for the formal description.

Lemma 6. *Let $v \in S_G$. G has a candidate elimination ordering if and only if $G - v$ has a candidate elimination ordering.*

Proof. Formally, we say that c is a candidate elimination ordering of G, if c is a permutation of the vertices of G. We define $c' = (c, v)$ to be the operation of appending v at the end of permutation c. Conversely, we define $c - v$ to be the operation of removing v from permutation c.

Let c be a candidate elimination ordering of $G - v$. Then, by definition of the set S_G, (c, v) is a candidate elimination ordering of G.

For the opposite direction, let c be a candidate elimination ordering of G. If v is the last vertex in c, then $c - v$ is trivially a candidate elimination ordering of $G - v$. Suppose that the last vertex of c is a vertex $u \neq v$. As $v \in S_G$, there exists a candidate parent w of v. If v does not give birth to any vertex in c then v is moved to the end of c, i.e., right after vertex u. Let c' be the resulting candidate elimination ordering of G; then $c' - v$ is a candidate elimination ordering of $G - v$, as the parent-child relations of $G - v$ are the same in both $c' - v$ and c.

Let v give birth to at least one vertex, and 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). This is always possible as $N[v] \subseteq N[w]$. Let now w appear after v in c, and $Z_0 = \{z \in Z : v <_c z <_c w\}$ 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). Again, 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), 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 $c''' - v$ is a candidate elimination ordering of $G - v$, as the parent-child relations of $G - v$ are the same in both $c''' - v$ and c''. $\qquad\square$

Theorem 1. *The* `candidate elimination ordering` *algorithm decides in polynomial time whether a connected graph G has a growth schedule of $n - 1$ slots and no excess edges, and outputs such a*

11

schedule if one exists.

Proof. It is easy to see by the description of the algorithm, that as long as there exists a candidate vertex in graph G in every iteration, the algorithm will output a growth schedule for G. What is left to show, is that we can greedily pick any candidate vertex and still output a growth schedule if one exists. This property is guaranteed by Lemma 6. Finally, for a connected graph G, we can find the candidate vertices in polynomial time, and thus, the algorithm terminates in polynomial time. $\qquad\square$

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 [25, 5, 33]. A cop-win graph is an undirected graph on which the pursuer (cop) can always win a pursuit–evasion game against a robber, with the players taking alternating turns in which they can choose to move along an edge of a graph or stay put, until the cop lands on the robber's vertex [11].

In fact, it is not hard to show that *a graph has a candidate elimination ordering if and only if it is a cop-win graph.* Due to this, our `candidate elimination ordering` algorithm might be similar to some folklore algorithms in the literature of cop-win graphs.

Lemma 7. *There is a modified version of the candidate elimination ordering algorithm that decides in polynomial time whether a connected graph G has a growth schedule of $n-1$ slots and ℓ excess edges, where ℓ is a constant, and outputs such a schedule if one 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 $x = 1, 2, \ldots, \frac{n(n-1)}{2} - |E|$ where each graph G'_x is a copy of G with the addition of one edge $e \notin E$, and we do this for all possible edge additions. In particular, we create $G'_x = (V'_x, E'_x)$, where $V'_x = V$ and $E'_x = E \cup \{uv\}$ such that $uv \notin E$ and $(E'_j \neq E'_i)$, for all $i \neq j$. Since the complement of G has at most $\frac{n(n-1)}{2}$ edges, we will create up to $\frac{n(n-1)}{2}$ graphs G'_x. 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 G. This process can be modified to work for any ℓ. As the number of graphs tested is at most $\frac{n^\ell(n-1)}{2}$, for constant ℓ the algorithm terminates in polynomial time. $\qquad\square$

The following algorithm decides whether a graph $G = (V, E)$ on n vertices has a growth schedule σ of $\log n$ slots and no excess edges, when $n = 2^\delta$, for some $\delta \geq 0$.

`Fast growth` algorithm: In every iteration t, the algorithm computes the set S_{G_t} of candidate vertices in G_t. It then tries to find a subset $L_t \subseteq S_{G_t}$ of candidate vertices that satisfy both of the following properties:

1. L_t is an independent set of $n/2$ vertices in graph G_t.

2. L_t contains candidate vertices of graph G_t

3. There is a perfect matching between the candidate vertices in L_t and the other vertices of graph G_t.

12

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

Assuming that we have a perfect matching M, for each edge $u_i v_i \in M$, the algorithm creates a variable x_i. The truthful assignment of x_i means that we pick v_i for L_t and the negative assignment means that we pick u_i for V_2. We add clauses to the 2-SAT formula ϕ as follows:

- If v_i is a candidate vertex and u_i is not, then has to be in $v_i \in L_t$, and so 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 $(\overline{x_i})$ to ϕ. If both u_i and v_i are candidate vertices, either one could be in L_t so we add clause $(x_i \vee \overline{x_i})$.

- We want L_t to be an independent set, so for each edge $u_i u_j \in E$, 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 L_t. Similarly, for every edge $v_i v_j \in E$, we add clause $(\overline{x_i}) \vee (\overline{x_j})$ to ϕ and for every edge $u_i v_j \in E$, we add clause $(x_i) \vee (\overline{x_j})$ to ϕ.

Lemma 8. *Let $G_t = (V, E)$ be a connected graph of n vertices. If G_t has a growth schedule of $\log n$ slots and of no excess edges then there exists a perfect matching M in G_t and a valid candidate vertex L_t, such that for every $u \in L_t$, there exists $uv \in M$, such that $v \notin L_t$.*

Proof. In order for a growth schedule σ to generate a graph G of n vertices in $\log n$ slots, in each slot t, every vertex in graph G_t must generate a vertex. Therefore, in the last slot of σ, there are $n/2$ vertices that generate $n/2$ vertices. Since the growth schedule has no excess edges, In the last slot, there are $n/2$ vertices for which $n/2$ other vertices are generated. Therefore, such a perfect matching M always exists where set L_t contains the children. $\square$

Lemma 9. *The 2-SAT formula ϕ, generated by the `fast growth` algorithm, has a solution if and only if there is a valid set of candidate vertices L_t in graph $G_t = (V, E)$.*

Proof. Let us assume that graph $G_t = (V, E)$ has a valid set of candidate vertices L_t. By Lemma 8, we also know that there is a perfect matching M between the vertices in L_t and the vertices in $V \setminus L_t$. By construction of the 2-SAT formula, each edge $u_i v_i \in M$ is represented by a variable x_i. The clauses added to the 2-SAT formula guarantee the following about any solution to it:

- if there is a set L_t, there are $n/2$ variables x_i created, and $n/2$ clauses of the form (x_i), $(\overline{x_i})$, and $(x_i \vee \overline{x_i})$ in formula ϕ, that can all be satisfied.

- Additionally, since L_t is an independent set, the clauses of the form $(x_i \vee x_j)$ can also be satisfied.

The inverse direction follows as well by construction of formula ϕ.

$\square$

Lemma 10. *Consider a connected graph $G_t = (V, E)$. If G_t has a growth schedule of $\log n$ slots and with no excess edges, then any perfect matching implies a valid candidate set $|L_t| = n/2$, where L_t has exactly one vertex for each edge of the perfect matching.*

13

Algorithm 2 Fast growth algorithm.

Input: A graph $G = (V, E)$ on $n = 2^\delta$ vertices.
Output: A growth schedule of $k = \log n$ slots and of no excess edges for G.

1: **for** $k = \log n$ downto 1 **do**
2:    $\mathcal{S}_k = \emptyset; \quad \phi = \emptyset$
3:    Find a perfect matching $M = \{u_i v_i : 1 \le i \le n/2\}$ of G.
4:    **if** No perfect matching exists **then**
5:       **return** "no"
6:    **for** every edge $u_i v_i \in M$ **do**
7:       Create variable x_i
8:    **for** every edge $u_i v_i \in M$ **do**
9:       **if** u_i is a candidate vertex and v_i is not **then**
10:         $\phi \leftarrow \phi \wedge (\overline{x_i})$
11:       **else if** u_i is a not candidate and v_i is a candidate **then**
12:         $\phi \leftarrow \phi \wedge (x_i)$
13:       **else if** u_i is not a candidate and v_i is not a candidate **then**
14:         **return** "no"
15:    **for** every edge $u_i u_j \in E \setminus M$ **do**
16:       $\phi \leftarrow \phi \wedge (x_i \vee x_j)$
17:    **for** every edge $v_i v_j \in E \setminus M$ **do**
18:       $\phi \leftarrow \phi \wedge (\overline{x_i} \vee \overline{x_j})$
19:    **for** every edge $u_i v_j \in E \setminus M$ **do**
20:       $\phi \leftarrow \phi \wedge (x_i \vee \overline{x_j})$
21:    **if** ϕ is satisfiable **then**
22:       Let τ be a satisfying truth assignment for ϕ
23:       **for** $i = 1, 2, \ldots, n/2$ **do**
24:         **if** $x_i = true$ in τ **then**
25:           $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup (u_i, v_i, \{v_i w : w \in N(v_i)\})$
26:           $V \leftarrow V \setminus \{v_i\}$
27:           $E \leftarrow E \setminus \{v_i w : w \in N(v_i)\}$
28:         **else** $\{x_i = false$ in $\tau\}$
29:           $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup (v_i, u_i, \{u_i w : w \in N(u_i)\})$
30:           $V \leftarrow V \setminus \{u_i\}$
31:           $E \leftarrow E \setminus \{u_i w : w \in N(u_i)\}$
32:    **else** $\{\phi$ is not satisfiable$\}$
33:       **return** "no"
34: **return** $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k, \emptyset)$

Proof. By Lemma 9, any perfect matching M contains edges uv, such that there exists a valid candidate set L_t that contains one vertex exactly for each edge $uv \in M$. Thus, if graph G_t has a growth schedule, the solution to the 2-SAT formula corresponds to a valid candidate set L_t. $\square$

Theorem 2. *For a connected 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 of no excess edges, if one exists.*

Proof. By Lemmas 9 and 10, 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 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. $\qquad\square$

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 4. *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.

Theorem 3. *The decision version of the zero-excess 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 grown 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 grow 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

15

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 $\kappa(G') \leq \chi(G) + n$. $\qquad\square$

Theorem 4. *Let $\varepsilon > 0$. If there exists a polynomial-time algorithm, which, for every connected graph G, computes a $n^{\frac{1}{3} - \varepsilon}$-approximate growth schedule (i.e., a growth schedule of at most $n^{\frac{1}{3} - \varepsilon} \kappa(G)$ slots), then $\mathrm{P} = \mathrm{NP}$.*

Proof. The reduction is from the minimum coloring problem. Given an arbitrary graph $G = (V, E)$ of n vertices, we grow in polynomial time a graph $G' = (V', E')$ of $N = 4n^3$ 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 $2n$ vertices, 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 in $\lceil \log n^2 \rceil$ slots and of no excess edges (see Lemma 3)[5]. This completes the construction of G'. Clearly, G' can be grown in time polynomial in n.

Now we will prove that there exists a growth schedule σ' of G' of number of slots 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 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 $N[u] \subseteq V(G_i^A) \cup V(G_i^B) \cup V(G_{i-1}^A) \cup$

---

[5] From Lemma 3 it follows that the path on n^2 vertices can be grown 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 $a_1, \ldots, a_{n^2}$.

$V(G_{i-1}^A) \cup V(C_i) \subseteq N[w]$. 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. 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

$$\kappa(G') \leq n^2 \chi(G) + 4n - 2 + \lceil 2 \log n \rceil. \tag{1}$$

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 $V(G_i^A)$ (respectively $V(G_i^B)$) which are generated in this slot are independent. For every $i = 1, \ldots, n^2$, denote by χ_i^A (respectively χ_i^B) the number of different slots of σ'' in which at least one vertex of $V(G_i^A)$ (respectively $V(G_i^B)$) 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,

$$k \geq \frac{1}{2} n^2 \chi^*. \tag{2}$$

Recall that $N = 4n^3$. It follows by Equations (1) and (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 [40]. $\qquad \square$

4. Growth Schedules of (Poly)logarithmic Slots

In this section, we study graphs that have growth schedules of (poly)logarithmic slots. As proved in Section 3, an integral factor in computing a growth schedule for any graph G, is computing a k-coloring for G. In particular, a growth schedule for graph G using k slots implies that graph G can be colored with k colors. 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.

4.1. Trees

We give 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 (from Section 2.2).

Tree **algorithm**: Starting from a tree graph G, the algorithm alternats 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 $u_1 u_\nu$ between the endpoints of P is activated and vertices $u_2, u_3, ...u_{\nu-1}$ are removed along with their incident edges. An example of this is shown in Figure 4. 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. An example of this is shown in Figure 5. If a single vertex is left, the algorithm terminates; otherwise, it proceeds to the path-cut phase.

The algorithm reverses the phases (by decreasing i) to output a growth schedule for the tree G as follows. For each path-cut phase, all path subgraphs that were decomposed in phase are regrown by using the *path* schedule as a sub-process. These can be executed in parallel in $O(\log n)$ slots. The same holds true for leaf-cut phases, 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.

Lemma 11. *Given any tree graph G, the algorithm deconstructs G into a single vertex using at most $2 \log n$ 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 $|N(u)| = 1$ and internal vertices $v \in S_v$ with $|N(v)| > 2$. 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 number of vertices 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. $\square$
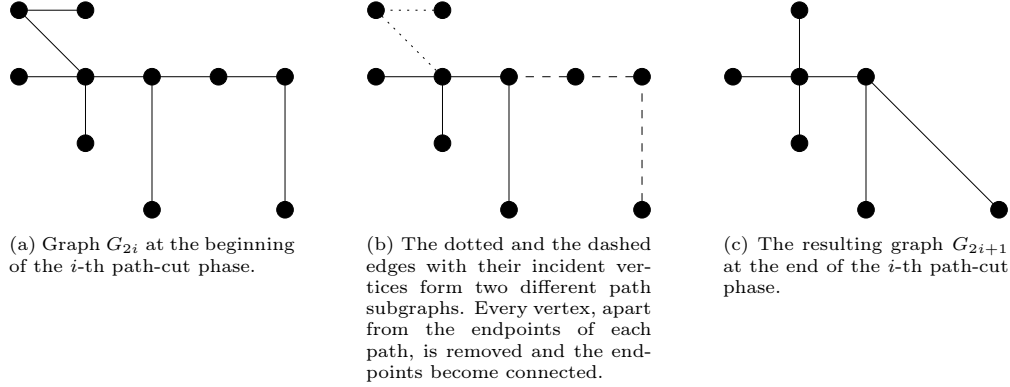
18

(a) Graph G_{2i} 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 4: An example of a path-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.

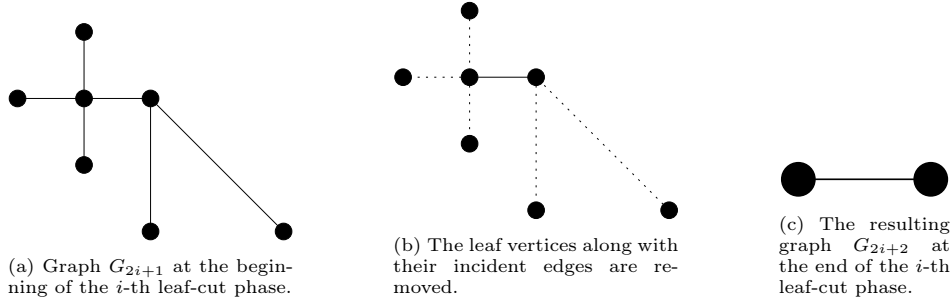(c) The resulting graph G_{2i+2} at the end of the i-th leaf-cut phase.

Figure 5: An example of a leaf-cut phase.

Lemma 12. *Every phase can be reversed using a growth schedule of $O(\log n)$ slots.*

Proof. 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 $u_1, u_2, \ldots u_x$ that we want to generate, we use vertex u_1 as the initiator and we execute the **path** algorithm from Section 2.2 in order to generate vertices $u_2, u_3, ..., u_{x-1}$. We make 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 $\log x \le \log n$ 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 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 $\log x \le \log n$ 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 $\log n$ slots since every schedule has distinct initiators and they can run in parallel. □

19

Theorem 5. *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 of the produced by each phase 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 schedule generates $n-1$ vertices, the excess edges activated throughout the schedule are $O(n)$. □

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 4-coloring of G and partitions the vertices into color sets V_i, $1 \leq i \leq 4$. The color sets are used to compute the growth schedule for G. The schedule contains four 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 [35], the pre-processing step computes a 4-coloring of the target planar graph G. This creates color sets $V_i \subseteq V$, where $1 \leq i \leq 4$, every color set V_i containing all vertices of color i. Without loss of generality, we can assume that $|V_1| \geq |V_2| \geq |V_3| \geq |V_4| \geq |V_4|$. 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, $1 \leq i \leq 4$, 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 activates $O(n)$ edges and (1) activates $O(n \log n)$ edges, and by observing that the schedule contains `star` sub-schedules that have $4 \times O(\log n)$ slots in total, the next lemma follows.

Lemma 13. *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 four sets V_i and we generate each set in a different phase i. This is always possible for any arbitrary 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 $wu_x \in E(G)$, 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 14. *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 grows V_i requires the same number of slots as `path`, which is $\lceil \log n_i \rceil$ slots. Combining the four sub-schedules requires $\sum_{i=1}^{4} \log n_i = \log \prod_{i=1}^{4} n_i < 4 \log n = O(\log n)$ 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 number of slots 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 uv 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 follows from Lemmas 13 and 14.

Theorem 6. *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 5. *A k-degenerate graph G is an undirected graph in which every subgraph has a vertex of degree at most k.*

Corollary 1. *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 14, 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 14, 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 14, the algorithm would produce a growth schedule of $O((\Delta + 1) \log n)$ slots. □

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} with n vertices, 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 7. *Any growth schedule σ of $\log n$ slots for 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 $uv \notin E(G)$. 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. $\square$

Corollary 2. *Any growth schedule of $\log n$ slots for 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. $\square$

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 at distance $d = 2$ 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 15. *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 8. *Consider graph $G_{bipart} = (V', E')$ of n vertices. 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 target graph to have n vertices. 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 $i = 1, 2, \ldots, n/2$ and $a_i, b_i \in E'$, 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 are generated in the last slot, only vertices from A or B, respectively, can be generated in that same slot. Thus, we can conclude that the last slot must either contain every vertex in A or every vertex in B.

Without loss of generality, 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 $a_l a_i \in (E' \setminus a_i b_j)$, $a_l a_i$ 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 $|N(b_j)| - 1$ 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 must activate $\Omega(n \log n) - n = \Omega(n \log n)$ excess edges. □

5. Edge-activation distances $d \neq 2$

For completeness, in this section we study edge-activation distances $d \neq 2$ and show that in this cases there are simple and efficient algorithms for finding growth schedules. We begin with some basic properties for the case where $d = 1$.

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

Proposition 4. *For $d = 1$, the shortest growth schedule σ of a path graph (respectively a star graph) on n vertices has $\lceil n/2 \rceil$ (respectively $n - 1$) slots.*

Proof. Let G be the path graph on n vertices. By definition of the model for $d = 1$, edges can only be activated during vertex generation, between the generated vertex and its parent. Thus, increasing the number of vertices of the path can only be achieved by generating one new vertex

at each of the endpoints of the path. The number of vertices 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 n vertices 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 of $n-1$ leaves. Increasing the number of vertices 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. $\square$

Proposition 5. *Let $d = 1$ and $G = (V, E)$ be a tree graph with diameter D. Then any growth schedule σ for G requires at least $\lceil D/2 \rceil$ slots.*

Proof. Consider a path p of length D. By Proposition 4, p requires a growth schedule of at least $\lceil D/2 \rceil$ slots. $\square$

Proposition 6. *Let $d = 1$ and $G = (V, E)$ be a tree graph with maximum degree Δ. Then any growth schedule σ for G requires at least Δ slots.*

Proof. Consider a vertex $u \in G$ with degree Δ and let $G' = (V', E')$ be a subgraph of G, such that $V' = N[u]$ and $E' = E(N[u])$. Notice that G' is a star graph of $\Delta + 1$ vertices. By Proposition 4, any growth schedule for G' has at least Δ slots. $\square$

Proposition 7. *Let $d = 1$. Consider a tree graph G and a growth schedule σ for it. Denote by G_t the graph grown by the end of slot t of σ. Then any vertex generated in slot t must be a leaf 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, vertex u must be a leaf. $\square$

We now provide an algorithm, called **_trimming_** (see Algorithm 3), that optimally solves the graph growth problem for $d = 1$. The algorithm follows a bottom-up approach for building the intended growth schedule $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k, \mathcal{E}) = (\mathcal{K}_k, \mathcal{K}_{k-1}, \ldots, \mathcal{K}_1, \emptyset)$. In every iteration t of the algorithm, the parent-child pairs of $\mathcal{K}_t$ are formed between leaves and their parents on the tree. The leaves that were included in a parent-child pair are removed and the algorithm repeats. The process continues until graph G has a single vertex left, which is set as the initiator. In the next theorem, we show that the algorithm outputs an optimum growth schedule with respect to the number of slots.

Theorem 9. *For $d = 1$ and for any tree graph G, the* `trimming` *algorithm computes in polynomial time a slot-optimal growth schedule for G.*

Proof. Let $\sigma = (\mathcal{S}_1, \ldots, \mathcal{S}_k, \emptyset)$ be the growth schedule obtained by the **_trimming_** algorithm on 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.

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 $|L_{k-\ell}| \neq |L'_{k'-\ell}|$.

Algorithm 3 Trimming algorithm, for $d = 1$.

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

1: $t \leftarrow 1$
2: **while** $V \neq \emptyset$ **do**
3:      $\mathcal{K}_t = \emptyset$
4:      **for** each leaf vertex $v \in V$ and its unique neighbor $u \in V$ **do**
5:          **if** u is not marked as a "parent in $\mathcal{K}_t$" **then**
6:             Mark u as a "parent in $\mathcal{K}_t$"
7:             $\mathcal{K}_t \leftarrow \mathcal{K}_t \cup \{(u, v, \{uv\})\}$
8:             $V \leftarrow V \setminus \{v\}$
9:      $t \leftarrow t + 1$
10: **return** $\sigma = (\mathcal{K}_t, \mathcal{K}_{t-1}, \ldots, \mathcal{K}_1, \emptyset)$

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} (respectively $G'_{k'-1}$) that is obtained by removing from G the leafs of L_k (respectively 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 number of slots. That is, σ is an optimum growth schedule. $\quad\square$

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 number of vertices of the graph.

Lemma 16. *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 16 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 17. *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.*

6. Conclusion and Open Problems

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 grown, 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 number of slots of such a schedule, unless P $=$ NP. For the special case of schedules of $\log n$ slots and of no 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 is 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 by any growth schedule for 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. We also gave some efficient algorithms, mostly for specific graph families, but there seems to be room for more positive results.

Finally. we could extend the model and study how much this changes our results. One approach is to consider whether we can grow directed graphs or graphs with weighted edges. For example,

we could consider a model where each vertex can activate edges that sum up to at most a fixed weight per slot. Another interesting approach is to study a combination of the growth dynamics of the present work and the edge-modification dynamics of [30], thus, allowing the activation of edges between vertices generated in past slots.

References

[1] 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.

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

[3] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37:1–37:25, 2007.

[4] John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Towards robust and efficient computation in dynamic peer-to-peer networks. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 551–569, 2012.

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

[6] Albert-Laszlo Barabasi and Reka Albert. Albert, r.: Emergence of scaling in random networks. science 286, 509-512. *Science*, 286:509–12, 11 1999. `doi:10.1126/science.286.5439.509`.

[7] 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.

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

[9] Béla Bollobás. *Random graphs*. Number 73 in Cambridge studies in advanced mathematics. Cambridge University Press, 2nd edition, 2001.

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

[11] Anthony Bonato and Richard J. Nowakowski. *The game of cops and robbers on graphs*, volume 61 of *Student Mathematical Library*. American Mathematical Society, Providence, RI, 2011.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] George B. Mertzios, Othon Michail, George Skretas, Paul G. Spirakis, and Michail Theofilatos. The complexity of growing a graph. In *Algorithmics of Wireless Networks*, pages 123–137. Springer Cham, 2022.

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

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

[29] 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.

[30] Othon Michail, George Skretas, and Paul G. Spirakis. Distributed computation and reconfiguration in actively dynamic networks. *Distributed Computing*, 35:185–206, 2022.

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

[32] 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.

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

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

[35] Neil Robertson, Daniel P. Sanders, Paul Seymour, and Robin Thomas. Efficiently four-coloring planar graphs. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 571–575. Association for Computing Machinery, 1996.

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

[37] 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.

[38] 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.

[39] 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.

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