

# The Complexity of Growing a Graph

George B. Mertzios<sup>\*†</sup>   Othon Michail<sup>‡</sup>   George Skretas<sup>§</sup>   Paul G. Spirakis<sup>¶</sup>  
Michail Theofilatos<sup>\*\*†</sup>

## Abstract

Motivated by biological processes, we introduce here the model of growing graphs, a new model of highly dynamic networks. Such networks have as nodes entities that can self-replicate and thus can expand the size of the network. This gives rise to the problem of creating a target network  $G$  starting from a single entity (node). To properly model this, we assume that every node  $u$  can generate at most one node  $v$  at every round (or time slot), and every generated node  $v$  can activate edges with other nodes only at the time of its birth, provided that these nodes are up to a small distance  $d$  away from  $v$ . We show that the most interesting case is when the distance is  $d = 2$ . Edge deletions are allowed at any time slot. This creates a natural balance between how fast (time) and how efficiently (number of deleted edges) a target network can be generated. Note that, if one demands a target network to be constructed in a small number of time slots, then this may not be possible unless excess edges are introduced in the process in order to facilitate the generation of edges of the target network. A central question here is, given a target network  $G$  of  $n$  nodes, can  $G$  be constructed in the model of growing graphs in at most  $k$  time slots and with at most  $\ell$  excess edges? We consider here both centralized and distributed algorithms for such questions (and also their computational complexity).

Our results include lower bounds based on properties of the target network and algorithms for general graph classes that try to balance speed and efficiency. We then show that the optimal number of time slots to construct an input target graph with zero-waste (i.e., no edge deletions allowed), is hard even to approximate within  $n^{1-\varepsilon}$ , for any  $\varepsilon > 0$ , unless  $P=NP$ . On the contrary, the question of the feasibility of constructing a given target graph in  $\log n$  time slots and zero-waste, can be answered in polynomial time. Finally, we initiate a discussion on possible extensions for this model for a distributed setting.

## 1 Introduction

**From Biology to Algorithms for Dynamic Networks.** Organisms have a remarkable ability to develop from a single cell, through a process known as embryogenesis. Embryogenesis appears to be a highly complicated process, which is not yet fully understood. It involves, among other things, a well-orchestrated cellular division and differentiation, controlled tissue growth, and morphogenesis,

---

\*Department of Computer Science, Durham University, UK. Email: [george.mertzios@durham.ac.uk](mailto:george.mertzios@durham.ac.uk)

†Supported by the EPSRC grant EP/P020372/1.

‡Department of Computer Science, University of Liverpool, UK. Email: [othon.michail@liverpool.ac.uk](mailto:othon.michail@liverpool.ac.uk)

§Department of Computer Science, University of Liverpool, UK. Email: [g.skretas@liverpool.ac.uk](mailto:g.skretas@liverpool.ac.uk)

¶Department of Computer Science, University of Liverpool, UK and Computer Engineering & Informatics Department, University of Patras, Greece. Email: [p.spirakis@liverpool.ac.uk](mailto:p.spirakis@liverpool.ac.uk)

‖Supported by the NeST initiative of the School of EEE and CS at the University of Liverpool and by the EPSRC grant EP/P02002X/1.

\*\*Department of Computer Science, University of Liverpool, UK. Email: [michail.theofilatos@liverpool.ac.uk](mailto:michail.theofilatos@liverpool.ac.uk)

††Supported by the Leverhulme Research Centre for Functional Materials Design.

all resulting from a construction program carefully designed by evolution and stored within the organism’s genetic code. These chemically implemented biological algorithms, whose execution results in impressive physical dynamics, have long been inspiring researchers towards two main objectives. One is to understand the algorithmic mechanisms of nature, as in Turing’s pioneering work on morphogenesis [49], and the other is to develop artificial systems resembling them, such as Von Neumann’s universal constructor [39].

In the past century, and quite naturally, computing research has primarily concerned itself with understanding the fundamental principles of numerical computation. This has led to the development of a beautiful theory of computation and its various subtheories including complexity theory, distributed computing, algorithmic theories of graphs, learning, and games. Driven by the realization of the fact that many natural processes are essentially algorithmic and by the advent of some low-cost and accessible technologies for programming and controlling various forms of matter, there is a growing recent interest in abstracting biological processes and formalizing their algorithmic principles and in inventing new algorithmic techniques suitable for the existing technologies. Prominent examples of biologically and chemically inspired algorithmic frameworks are brain computation [30, 41, 50], ant colony optimization [13, 16, 19], population protocols [3, 38], DNA self-assembly [17, 45, 46, 52], the algorithmic theory of programmable matter [1, 22, 25, 36], robotics [31, 42, 47], and dynamically growing networks [27].

At the same time, there is an ongoing effort to set the algorithmic foundations of dynamic networks. This has started with passively dynamic networks, such as population protocols [3], other dynamic distributed computing models [28, 40], and the algorithmic and structural properties of temporal graphs [2, 6, 8, 18, 26, 33, 35, 53] and has been recently expanding to the investigation of actively dynamic networks, ranging from geometric reconfiguration models [14, 15, 20, 36] to abstract reconfigurable networks [4, 37] and even hybrid models [21, 24, 38].

Motivated by this progress and by the algorithmic principles of biological development, our goal here is to study an abstraction of networked systems which, starting from a single entity, can grow into well-defined global networks and structures. An interesting recent attempt in this direction was the Nubot model of Woods *et al.* [51], showing how to efficiently self-assemble shapes and patterns from simple monomers in a 2D discrete geometric environment. Their universal constructions that rely on stronger and/or global reconfigurations, e.g., via linear-strength actuation mechanisms, are beyond the scope of the present study. In particular, starting from their basic processes, we completely disregard geometry and lift growth and formation to the abstract graph or network level. This is motivated by the fact that any such biological or artificial dynamic networked system has an underlying dynamic graph as a reasonable representation of its dynamics. The underlying system that we aim to formalize in this work can be described as follows.

**Modeling Growing Networks.** A *growing graph* is modeled as an undirected dynamic graph  $G_t = (V_t, E_t)$  for  $t = 1, 2, \dots, k$ . The initial graph  $G_0$  consists only of a single node (singleton), called the *initiator* throughout the paper (i.e.,  $|V_0| = 1$ ). During the execution, the graph is gradually changing through the addition of nodes and edges and/or deletion of edges. Let  $u$  be a node that at time  $t$  generates a node  $w$ . We say that  $u$  is the *parent* of  $w$  and that  $w$  is the *child* of  $u$ ; we write  $u \xrightarrow{t} w$ . Each newly generated child  $w$  is initially connected (by default) to its parent, and can additionally activate some edges, only at the the time of its birth  $t$ , with any node  $v$  such that the distance between  $w$  and  $v$  in the “intermediate” graph  $(V_t, E_{t-1} \cup \{uw : u \xrightarrow{t} w\})$  is at most  $d$ . We call  $d$  the *edge-activation distance*. We only allow nodes to activate edges at the time of their birth because we want to model biological systems, where entities cannot receive large amounts of information to establish new connections but each entity can receive the information its parent has (via replication at the moment of its birth). The graph operations that our model

allows are the following:

- (i) *Node generation.* At each round  $t$ , each node of  $V_{t-1}$  can either decide to generate a new node, or not. Let  $V'_t \subseteq V_t$  be the set of nodes that decided to generate new nodes at time  $t$ . We denote  $V_t^+$  the set of nodes that were generated at time  $t$ .
- (ii) *Edge activation.* Each node  $u \in V_t^+$ , in addition to the edge with its parent, can also activate edges with nodes that are within its *edge-activation distance* at the time of its birth, including nodes generated in the same round. All edges remain activated throughout the execution of the algorithm, or until some node decides to remove an adjacent edge from the graph. We denote  $E_t^+$  the set of edges that were activated at time  $t$ .
- (iii) *Edge deletion.* In each round, any node can decide to remove some of its edges. We denote  $E_t^-$  the set of edges that are removed at time  $t$ .

At each round  $t \geq 0$  the graph  $G_{t+1} = (V_{t+1}, E_{t+1})$  is obtained from  $G_t = (V_t, E_t)$  as follows:  $V_{t+1} = V_t \cup V_t^+$ , and  $E_{t+1} = (E_t \setminus E_t^-) \cup E_t^+$ . Let  $u_0$  be the unique node of the initial graph  $G_0$ , and let  $w$  be a node that is born in round  $t$ , for some  $0 \leq t \leq k$ . The *birth path* of node  $w$  is the unique node sequence  $B_w = (u_0, u_{i_1}, \dots, u_{i_{p-1}}, u_{i_p} = w)$ , where  $i_p = t$  and  $u_{i_{j-1}} \xrightarrow{i_j} u_{i_j}$ , for every  $j = 1, 2, \dots, p$ . That is,  $B_w$  is the sequence of nodes which resulted in the birth of node  $w$ . Furthermore, the *progeny* of a node  $u$  is the set  $P_u$  of descendants of  $u$ , i.e.,  $P_u$  contains those nodes  $v$  such that  $u \in B_v$ .

The problem that we examine is the construction of a given connected graph  $G$ , starting from  $G_0$ . Throughout the paper, we call  $G$  the *target* graph. Note that if there is no bound on the number of *rounds* (or *time slots*) or on  $d$ , then for any target graph  $G$  there is a growing graph ending at  $G_k = G$ . Observe first that in the  $d = 2$  case, all nodes that are born within the same round  $i$  must forever form an independent set and that a newly born node in round  $i$ , apart from its parent, can only connect to the neighbors of its parent in  $G_{i-1}$ . There is a straightforward strategy that can generate any target graph  $G$  while respecting these constraints. The strategy assumes an ordering  $u_1, u_2, \dots, u_n$  of the nodes of  $G$ , where  $u_1$  is a maximum-degree node of  $G$  and will be the node mapped to the initial node of the growing graph. In every round  $i$ , for  $2 \leq i \leq n$ ,  $u_1$  gives birth to node  $u_i$ . Then, the edges  $u_1 u_i$  and  $u_i u_j$ , for  $j < i$  and  $u_i u_j \in E(G)$ , are being activated. Just before the end of round  $n$ ,  $u_1$  deactivates all edges  $u_1 u_i$  for which  $u_1 u_i \notin E(G)$  holds. This strategy constructs any target graph  $G$  in  $n - 1$  rounds while deactivating  $n - d_{\max}$  edges, where  $d_{\max}$  is the maximum degree of a node in  $G$ . We refer to such edges which are not in  $E(G)$ , but still need to be activated and later deactivated by a strategy, as *excess edges* for this strategy. Even though the strategy described above for  $d = 2$  is universal and quite efficient w.r.t. its number of excess edges, it is still considered here as highly inefficient w.r.t. its “round complexity” (i.e., the number of rounds needed to construct the target graph  $G$ ). In particular, in this work, a strategy is considered as efficient if it can achieve a round complexity which is (poly)logarithmic in the order of  $G$ , i.e., in  $n = |V(G)|$ . In fact, there are target graphs that do not admit an efficient construction strategy. For example, the complete graph  $K_n$  cannot be constructed in less than  $n - 1$  rounds, as otherwise  $K_n$  would have at least one independent set of size 2. Thus, we will now define the core problem of our paper for  $d = 2$ .

**Definition 1** (construction schedule for  $d = 2$ ). *Let  $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k)$  be an ordered sequence of sets, where each  $\mathcal{S}_i = \{(u_1, v_1, E_1), (u_2, v_2, E_2), \dots, (u_\ell, v_\ell, E_\ell)\}$  is an unordered set of ordered tuples  $\{(u_j, v_j, E_j) : 1 \leq j \leq \ell\}$  such that, for every  $j$ ,  $u_j$  and  $v_j$  are nodes (where  $u_j$  gives birth to*

$v_j$ ) and  $E_j$  is a set of edges incident to  $v_j$  such that  $u_j v_j \in E_j$ . Suppose that, for every  $2 \leq i \leq k$ , the following conditions are all satisfied:

- each of the sets  $\{v_1, v_2, \dots, v_\ell\}$  and  $\{u_1, u_2, \dots, u_\ell\}$  contains  $\ell$  distinct nodes,
- each node  $v_j \in \{v_1, v_2, \dots, v_\ell\}$  does not appear in any set among  $\mathcal{S}_1, \dots, \mathcal{S}_{i-1}$  (i.e.,  $v_j$  is “born” at time slot  $i$ ),
- for each node  $u_j \in \{u_1, u_2, \dots, u_\ell\}$ , there exists exactly one set among  $\mathcal{S}_1, \dots, \mathcal{S}_{i-1}$  which contains a tuple  $(u', u_j, E')$  (i.e.,  $u_j$  was “born” at a time slot before slot  $i$ ).

Let  $i \in \{2, \dots, k\}$ , and let  $u$  be a node that has been generated at some time slot  $i' \leq i$ , that is,  $u$  appears in at least one tuple of a set among  $\mathcal{S}_1, \dots, \mathcal{S}_i$ . We denote by  $E^i$  the union of all edge sets that appear in the tuples of the sets  $\mathcal{S}_1, \dots, \mathcal{S}_i$ ;  $E^i$  is the set of all edges generated until time slot  $i$ . We denote by  $E^i(u) \subseteq E^i$  the set of edges of  $E^i$  which are incident to node  $u$ . Furthermore, we denote by  $N_i(u)$  the set of neighbors of  $u$  in the set  $E^i(u)$ . If, in addition, for every  $2 \leq i \leq k$  and for every tuple  $(u_j, v_j, E_j) \in \mathcal{S}_i$  we have that  $N_i(v_j) \subseteq N_i(u_j)$ , then  $\sigma$  is a construction schedule. The number  $k$  of sets in  $\sigma$  is the length of  $\sigma$ .

Moreover, let  $V$  be the set of all nodes which appear at at least one tuple in  $\sigma$ , and let  $E = \bigcup_{u \in V} E^k(u)$  be the union of all edge sets of the tuples of  $\sigma$ . Then we say that the graph  $G = (V, E)$  is constructed by the construction schedule  $\sigma$  with  $k$  slots (where  $k$  is the number of sets  $\sigma$ ). If a graph  $G' = (V, E')$  is obtained by removing at most  $\ell$  edges from  $G$ , then we say that  $G'$  is constructed by  $\sigma$  with  $k$  slots and at most  $\ell$  excess edges.

**Construction Schedule Problem:** Given a target graph  $G$ , compute in polynomial time a construction schedule of length at most  $k$  with at most  $\ell$  excess edges, if it exists.

To the best of our knowledge, this is the first study on this model and problems. Due to this, our primary focus is on understanding the centralized complexity of the problems as well as the structural properties and the dynamics of growing graphs.

**Our Contribution.** Since this model can potentially be used in multiple fields as discussed in the introduction, the paper focuses solely on the fundamental case of centralized and deterministic algorithms. The main focus of our results is for  $d = 2$ . There are mainly two reasons for that. First, the case of  $d = 2$  is more natural both for real systems, where a generated node is usually able to communicate only with nodes who are close to it at the time of its birth, and for more abstract systems, where a generated node learns the id of the neighbors of its parent node (through the parent node) and connects with them. Secondly, the cases  $d = 1$  and  $d \geq 3$  admit very simple and efficient construction schedules which are formally shown in Section 2.1. In Section 2.2, we provide some basic propositions that are crucial to understanding the limitations on the speed/efficiency of a construction schedule of a graph  $G$ . We then use these propositions to provide some lower bounds.

In Section 3, we begin with our algorithmic constructions with some basic algorithms for special types of graphs which will then be used as core components for our more general algorithms. Here we provide an algorithm that outputs a construction schedule for a tree graph  $G$ , with  $O(\log^2 n)$  slots but only  $O(n)$  excess edges, and an algorithm that outputs a construction schedule for any planar graph, with  $O(\log n)$  slots and  $O(n \log n)$  excess edges.

In Section 4, our goal is to study graphs whose construction schedules are very efficient. Here, we introduce the zero-waste construction schedule problem, where the goal is to decide whether a graph  $G$  admits a construction with  $k$  slots and  $\ell = 0$  excess edges. Our main technical contribution

here consists of two hardness results. First, we show that the decision version of the zero-waste construction schedule is NP-complete. Second, we show that, unless  $P=NP$ , there is no polynomial-time algorithm which, for every graph  $G$  computes a  $n^{1-\epsilon}$ -approximate construction schedule. On the positive side, we first show that a graph  $G$  has a construction schedule with  $k = n - 1$  slots and  $\ell = 0$  excess edges if and only if it is a *cop-win* graph. Cop-win graphs have been studied extensively in the past [5,29,43]. We then proceed to our main positive result, where we provide an algorithm that computes whether a graph  $G = (V, E)$ ,  $|V| = 2^\delta$  has a construction schedule with  $k = \log n$  slots and  $\ell = 0$  excess edges. If it does, the algorithm also outputs such a construction schedule.

In Section 5 we conclude with some open problems on this model and we initiate a discussion on the distributed construction of graphs. We start by providing a distributed version of the model, where the nodes operate autonomously and have limited capabilities, and we then provide protocols for the distributed construction of cycles and 4-regular graphs with a Hamiltonian cycle. Finally we discuss other possible extensions of the model that better represent real systems.

## Related Work

**Programmable Matter and Self-Assembly.** Programmable matter refers to any type of matter that can be programmed to change its physical properties, such as its shape [22]. Progress in small-scale engineering has enabled the production of tiny monads, whose size ranges from milli down to nano and which are equipped with computation, communication, sensing, and actuation capabilities. These monads are bound to neighboring monads, usually by electromagnetic or electrostatic forces, forming a connected 2D or 3D shape and may be programmed to reconfigure in order to adapt to their environment or to solve a task requiring modification of their joint physical properties. Equally impressive progress has been recently made in the domain of DNA self-assembly. There, DNA strands have been successfully programmed to self-assemble into any desired nano-scale pattern [45], to implement registers or to simulate circuits [52] and the goal of universally programming molecules seems now to be within reach. Both directions are based on and further inspiring the development of solid algorithmic foundations [1,14,17,36,46]. Our model may be viewed as an network-level abstraction of programmable matter systems and models in which the individual monads can self-replicate or introduce to the system new monads from a pool. Such a model, is the one by Woods *et al.* [51]. Even though it has offered an inspiration for our work, our model completely disregards geometry as we are aiming to develop a more general framework for networked systems that grow through self-replication and local reconfiguration.

**Temporal Graphs.** A temporal graph is a graph that changes with time [35]. Starting with the works of Berman [6] and Kempe *et al.* [26] in the single-labeled case, in which every edge of an underlying graph can be available at most once, and continuing with subsequent work on the multi-labeled case [18,33,53], a temporal extension of the algorithmic principles of graph theory is currently under development. The vast majority of temporal graphs considered in the literature have a static set of nodes and the dynamics concern only the edges. Our model of growing graphs is a type of a temporally changing graph in which the node set is non-decreasing, and typically strictly increasing, while the edges follow some locally and temporally constrained dynamics. In particular, an edge can only be activated upon the birth of its latest-born endpoint, then remains constantly active for one or more rounds, and can be deactivated at most once. With respect to this domain, we believe that our work might inspire further investigation of temporal graphs with a dynamically changing set of nodes.

**Distributed Computation in Dynamic Networks.** One way of classifying dynamic networks

is on who controls the dynamics. When control is passive, the distributed entities cannot control their interaction pattern as this is exclusively determined by the environment in which they reside. Probably the first such model to be considered was the population protocol model of Angluin *et al.* [3], in which finite-state automata interact randomly in pairs. This was followed by models of more powerful devices that can communicate over general networks which are dynamically changing from round to round [28,40]. When, on the other hand, control is active, then the network dynamics are generated by the devices themselves and the goal is typically to efficiently reach a good target network when starting from an arbitrary initial connected network [4,24,37]. Hybrid control has also been considered [21,38]. The model considered here has been partially inspired by the approach of [37], where the geometric dynamics of programmable matter systems were lifted to the network level to study fast (i.e., polylog) reconfiguration of networks from any initial connected network to a small diameter (i.e., polylog) target network. Similarly, we here consider a type of actively dynamic network as an abstraction of biological and artificial systems growing in polylog time. In contrast to [37], we mostly explore here the centralized complexity of the underlying transformation problems, even though we also give some first distributed algorithms. We mostly leave the distributed case as an interesting direction for future research.

**Graph Elimination Orderings.** Many graph problems can be algorithmically solved by exploiting the properties of specific orderings of their nodes. Many times such node orderings can be computed by an appropriate *graph searching* algorithm, the most classical and well-known node orderings being the *Breadth-First Search (BFS)* and *Depth-First Search (DFS)* algorithms. A graph search algorithm provides a strategy for determining the next node to visit at every step. Although BFS and DFS solve many fundamental problems such as connectivity problems, various extensions of them are known for solving more specialized problems on graphs and networks. Arguably one the most famous and classical example is the problem of computing a perfect elimination ordering for a chordal graph. Since the mid 70s, it is known that chordal graphs can be efficiently recognized (and a perfect elimination ordering of them can be produced) by a variation of Breadth-First Search (BFS), namely the Lexicographic BFS (or Lex-BFS for short), due to Rose, Tarjan, and Lueker [44]. Since then, considerable progress has been made in developing new graph searching algorithms such as the Maximum Cardinality Search of Tarjan and Yannakakis [7,48], the Maximum Neighborhood Search [11], and the Lexicographic DFS [10,12,32,34], see also [11] for an overview. a distance-preserving elimination ordering and a domination elimination ordering [9].

## 2 Preliminaries

Given a connected undirected graph  $G = (V, E)$ , an edge between two nodes  $u$  and  $v$  is denoted by  $uv$ , and in this case  $u$  and  $v$  are *adjacent*. The open (resp. closed) neighborhood of a node  $u$  is the set  $N_G(u) = \{v \in V : uv \in E\}$  (resp.  $N_G[u] = N_G(u) \cup \{u\}$ ). Whenever  $G$  is clear from the context, we will refer to these sets as  $N(u)$  and  $N[u]$ , respectively. For any graph-theoretic notation that is not explicitly defined here, we refer the reader to [23].

### 2.1 The case $d = 1$ and $d \geq 3$

In this section, we show that for edge-activation distance  $d = 1$  or  $d \geq 3$  there are simple but very efficient algorithms for finding construction schedules. After that, the rest of the paper focuses on the case for  $d = 2$ . As a warm-up, we begin with a simple observation for the special case where the edge activation distance  $d$  is equal to 1.

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

**Proposition 1.** For  $d = 1$ , the shortest construction schedule of a line graph (resp. a star graph) with  $n$  nodes has  $\lceil n/2 \rceil$  (resp.  $n - 1$ ) slots.

*Proof.* Let  $G$  be the line graph with  $n$  nodes. By definition of the model, increasing the size of the line can only be achieved by generating one new node at each of the endpoints of the line. Then, the size of a line can only be increased by 2 in each slot after the first one. Therefore, in order to create any line graph of size  $n$ , starting from a single node, would require at least  $\lceil n/2 \rceil$  slots. The construction schedule where one node is generated at each of the endpoints of the line in each slot creates the line graph of  $n$  nodes in  $\lceil n/2 \rceil$  slots.

Now let  $G$  be the star graph with  $n - 1$  leaves. Increasing the size of the star graph can only be achieved by having the center node generate a new leaf, and this can occur at most once per slot. Therefore, the construction of  $G$  requires exactly  $n - 1$  slots.  $\square$

**Observation 2.** Let  $d = 1$  and  $G = (V, E)$  be a graph with diameter  $diam$ . Then any schedule that constructs  $G$  requires at least  $\lceil diam/2 \rceil$  slots to complete the construction.

*Proof.* Consider a path  $p$  of size  $diam$  that realizes the diameter of graph  $G$ . By Proposition 1 we know that  $p$  alone has construction time at least  $\lceil diam/2 \rceil$ .  $\square$

**Observation 3.** Let  $d = 1$  and  $G = (V, E)$  be a graph with maximum degree  $deg$ . Then any schedule that constructs  $G$  requires at least  $deg$  slots to complete the construction.

*Proof.* Consider a node  $u \in G$  with degree  $deg$  and let  $G' = (V', E')$  be a subgraph of  $G$ , such that  $V' = N_G[u]$  and  $E' = E(N_G[u])$ . Notice that  $G'$  is a star graph. By Proposition 1, we know that the construction schedule of  $G'$  has at least  $k$  slots.  $\square$

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

**Observation 4.** Let  $d = 1$ . Consider a graph  $G$  and a construction schedule  $\sigma$  for graph  $G$ . Consider a graph  $G_t$  to be the graph constructed at the end of slot  $t$  of  $\sigma$ . Then any node generated in slot  $t$  must be a leaf node in  $G_t$ .

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

The algorithm works as follows. Starting from graph  $G$ , in every round  $i$ , the algorithm finds an arbitrary maximum set of nodes  $L$  that could have been generated in the last slot of any construction schedule  $\sigma'$ . This is done by going through every leaf node  $u$  and its neighbor  $v$  of  $G$  and adding the tuple  $(v, u, uv)$  to slot  $S_i$  of our construction schedule  $\sigma$ , if no other tuple in  $S_i$  contains node  $v$ . The set  $L$  is the set of nodes generated in  $S_i$ . After that, we remove  $L$  from  $G$  and move to the next round. The process is repeated until graph  $G$  has a single node left which is added in the first slot of  $\sigma$  as the initiator. See Algorithm 1. We will prove that the *trimming* algorithm is optimal by showing that choosing an arbitrary maximum set of nodes  $L$  to generate in any slot  $t$ , maximizes  $|L + L'|$  where  $L'$  is an arbitrary maximum set of nodes that can be generated in graph  $(G - L)$ . This means that choosing any arbitrary maximum set  $L$  is always optimal.

**Lemma 1.** Let  $d = 1$ . Consider a graph  $G$  and a construction schedule  $\sigma$  for  $G$ , and let  $S_G$  be the set of leaf nodes in  $G$ . Consider  $L_1 \subseteq S_G$  to be an arbitrary maximum set of nodes generated in slot  $t$  of  $G$  and  $L_2 \subseteq S_G$  be another possible arbitrary set of nodes generated in slot  $t$  of  $G$  such

---

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

---

**Input:** A tree graph  $G = (V, E)$  with  $n$  nodes.

**Output:** An optimal construction schedule for  $G$ .

```

1:  $k \leftarrow 1$ 
2: while  $V \neq \emptyset$  do
3:    $\mathcal{S}_k = \emptyset$ 
4:   for each leaf node  $v \in V$  and its unique neighbor  $u \in V$  do
5:     Mark  $u$  as a “parent in  $\mathcal{S}_k$ ”
6:     if  $u$  is not marked as a parent in  $\mathcal{S}_k$  then
7:        $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup \{(u, v, uv)\}$ 
8:        $V \leftarrow V \setminus \{v\}$ 
9:        $k \leftarrow k + 1$ 
10:  if  $V = \emptyset$  then
11:    return  $\sigma = (\mathcal{S}_k, \mathcal{S}_{k-1}, \dots, \mathcal{S}_1)$ 

```

---

that  $|L_1| \geq |L_2|$ . Let graph  $(G - L)$  be the graph after removing set  $L$  from  $G$ . Then consider any arbitrary maximum sets of nodes  $L'_1, L'_2$  generated in the slot  $t - 1$  for graphs  $(G - L_1)$  and  $(G - L_2)$ , respectively. Then,  $|L_1 + L'_1| \geq |L_2 + L'_2|$ .

*Proof.* Consider any two arbitrary maximum sets  $L_1, L_2$  of nodes to be generated in slot  $t$  in graph  $G$ . Observe that the intersection of  $L_1, L_2$  contains the leaf nodes  $u$  whose neighbor  $v$  has degree equal to 2, since  $v$  has only one leaf node neighbor, which is  $u$ . The difference in the sets stems from leaf nodes  $u_1, u_2, \dots, u_x$  for  $x \leq n - 1$ , that share a common neighbor. Both sets  $L_i, L_j$  have exactly one node from  $u_1, u_2, \dots, u_x$  but not necessarily the same. Therefore,  $|L_1| = |L_2|$  and  $(G - L_1)$  and  $(G - L_2)$  contain the same number of leaf nodes. Thus, all maximum sets  $L'_1, L'_2$  in slot  $t - 1$  are equal in size for graphs  $(G - L_1)$  and  $(G - L_2)$ .

Assume now that  $L_1 > L_2$ ,  $L_1$  is a arbitrary maximum set, and consider  $L'_1, L'_2$  to be two arbitrary maximum sets in slot  $t - 1$  for  $(G - L_1)$  and  $(G - L_2)$ , respectively.

For any leaf node  $u \in V(G)$  whose neighbor has degree equal to 2, if  $u \in L_1$  and  $u \notin L_2$ , then in graph  $(G - L_2)$ ,  $u \in L'_2$ . This means that each node  $u$  is contained in  $L_1 \cup L'_1$  and  $L_2 \cup L'_2$ .

For each set of leaf nodes  $u_1, u_2, \dots, u_x$  for  $x \leq n - 1$ , that share a common neighbor in  $G$ , set  $L_1$  has one node  $u_i$  from such set  $u_1, u_2, \dots, u_x$ , while  $L_2$  has at most one node  $u_i$ . From that same set in  $(G - L_1)$  and  $(G - L_2)$ ,  $L'_1$  has one node  $u_i$  and  $L'_2$  has one node  $u_j$ . Therefore, for each set that contains  $u_1, u_2, \dots, u_x$ ,  $L_1 \cup L'_1$  contains two nodes, and  $L_2 \cup L'_2$  at most two nodes.  $\square$

**Theorem 1.** For  $d = 1$ , the trimming algorithm computes in polynomial time an optimal (shortest) construction schedule with  $\kappa$  slots for any tree graph  $G$ .

*Proof.* In every round, the algorithm computes an arbitrary maximum number of nodes that can be generated in the last slot  $t$  of graph  $G$  and then removes these nodes from  $G$ . By Lemma 1, doing this in slot  $t$ , maximizes the number of nodes generated in slots  $t$  and  $t - 1$ . By induction, this means that the set of slots created by the algorithm are a valid construction schedule with the minimum number of slots possible for graph  $G$ .  $\square$

We now 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 construction schedule with an optimal number of slots and only linear to the size of the graph excess edges.

**Lemma 2.** For  $d \geq 4$ , a graph  $G = (V, E)$  with  $n$  nodes can be generated with a construction schedule with  $\lceil \log n \rceil$  slots and with  $O(n)$  excess edges.

*Proof.* Let  $G_f = (V_f, E_f)$  be the target graph, and  $G_t = (V_t, E_t)$  be the constructed graph at time  $t$ . Initially  $V_0 = \{u\}$ , and  $E_0 = \emptyset$ . The goal is that  $V = V_{\lceil \log n \rceil}$  and  $E \subseteq E_{\lceil \log n \rceil}$ .

When a node  $w$  is generated, it is matched with an unmatched node of  $G$ . For any pair of nodes  $v, w \in G_{\lceil \log n \rceil}$  that have been matched with a pair of nodes  $v_f, w_f \in G_f$ , respectively, if  $(v_f, w_f) \in E_f$ , then  $(v, w) \in E_{\lceil \log n \rceil}$ , and if  $(v_f, w_f) \notin E_f$ , then  $(v, w) \notin E_{\lceil \log n \rceil}$ .

To achieve a construction of  $G_f$  in  $\lceil \log n \rceil$  slots, each node of  $G_t$  must generate a new node at slot  $t + 1$ , except possibly for the last slot of the construction schedule. To prove the lemma, we show that the construction 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 removed until slot  $\lceil \log n \rceil$ . The children of all leaves of the star are in distance 2 from  $u$ , therefore they activate their edge between them and  $u$  at the time of their birth.

The above construction means that the distance of any two nodes is always less or equal to four. Therefore, each node  $w$  that is generated at time  $t$  and is matched to a node  $w_f \in G_f$ , it activates the edges with each node  $u$  that has been generated and matched to node  $u_f \in G_f$  and  $(w_f, u_f) \in E_f$ . 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 nodes). Any other edge is activated only if it exists in  $G_f$ .  $\square$

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

**Lemma 3.** For  $d \geq 3$ , a graph  $G = (V, E)$  with  $n$  nodes can be generated with a construction schedule with  $\lceil \log n \rceil$  slots and with  $O(n^2)$  excess edges.

## 2.2 First remarks on $d = 2$

For the rest of the paper, we always assume that  $d = 2$ . In this section, we provide insight for  $d = 2$  by showing some basic properties of each graph  $G$  which restrict the possible construction schedules and we also provide some lower bounds. In the next proposition we show that the nodes generated in each time slot form an independent set in the constructed graph, i.e., any pair of nodes generated in the same slot cannot have an edge between them in the final graph.

**Proposition 2.** The nodes generated in a time slot form an independent set in the final graph.

*Proof.* Let  $G_t$  be our graph at time slot  $t$ . By definition of our model, any newly generated nodes are only allowed to activate edges within their *edge activation distance* at the time of their birth  $t$  with nodes in  $G_t$ . Consider any pair of nodes  $u_1, u_2$  that have minimal distance between them, in other words, they are neighbors and  $dist = 1$ . Assume that nodes  $u_1, u_2$  generate new nodes  $v_1, v_2$  at time slot  $t$ , respectively. The distance between nodes  $v_1, v_2$  at time slot  $t$  is  $dist = 3$  and therefore they cannot activate an edge between them. Finally, for any other pair of non-neighboring nodes, the distance between their children is  $dist > 3$ , thus remaining an independent set.  $\square$

**Proposition 3.** Consider any construction schedule  $\sigma$  for graph  $G$ . Let  $t_1, t_2$ ,  $t_1 \leq t_2$ , be the slots in which a pair of nodes  $u, w$  is generated, respectively. Let  $dist$  be the distance between  $u$  and  $w$  at time  $t_2$ . Then, at any time  $t \geq t_2$ , the distance between  $u$  and  $w$  can never become less than  $dist$ .

*Proof.* Given that the *edge activation distance* is two means that any node that is generated at slot  $t$ , can activate edges only with its parent and with the neighbors of its parent.

Let  $u$  be a node that is generated by a node  $u'$  at  $t_1$ , and  $w$  be a node that is generated by a node  $w'$  at  $t_2$ . Let  $G'$  be the graph at slot  $t_2$ , and  $P$ ,  $|P| = d$ , be the shortest path between  $u$  and  $w$  in  $G'$ . We distinguish two cases:

1. Node  $u$  and/or  $w$  generate new node(s), that are connected to all neighbors of  $u$  and/or  $w$ . In this case, the path that contains the new node(s) will clearly be larger than  $d$ .
2. In this case, some node  $p$  in the path between  $u$  and  $w$  generates a new node  $p'$  and activates all edges with the neighbors of  $p$ . In this case, the path that passes through  $p'$  will clearly have the same size as  $P$ .

It is then obvious that no construction schedule starting from  $G'$  can reduce the shortest distance between  $u$  and  $w$ . □

**Proposition 4.** *Consider  $t_1, t_2$ , where  $t_1 \leq t_2$ , to be the slots in which a pair of nodes  $u, w$  is generated, respectively, and edge  $(u, w)$  is not activated at  $t_2$ . Then any pair of nodes  $v, z$  cannot be neighbors if  $u \in B_v$  and  $w \in B_z$ .*

*Proof.* Given that the nodes  $u$  and  $w$  do not activate the edge between them, and by Proposition 3, the children of  $u$  will always be in distance at least 2 from  $w$  (i.e., they can activate edges only with the nodes that belong to the neighborhood of their parent node, and no edge insertions can reduce their distance). Sequentially, the same holds also for the children of  $w$ . All nodes that belong to the progeny  $P_u$  of  $u$  (i.e., each node  $z$  such that  $u \in B_z$ ) have to be in distance at least 2 from  $w$ , therefore they cannot be neighbors with any node in  $P_w$ . □

We will now provide some lower bounds on the number of slots for any construction schedule  $\sigma$  for graph  $G$ .

**Lemma 4.** *Assume that graph  $G$  has chromatic number  $\chi(G)$ . Then any schedule that constructs  $G$  requires at least  $\chi(G)$  slots.*

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

**Lemma 5.** *Assume that graph  $G$  has a clique  $q$  of size  $c$ . Then any valid construction schedule for  $G$  requires at least  $c$  slots.*

*Proof.* By Proposition 2, we know that every slot must contain an independent set of the graph and cannot contain more than one node from clique  $q$ . By the pigeon hole principle, it follows that  $\sigma$  must have at least  $c$  slots. □

### 3 Algorithms for Basic Graph Classes

In this section, we are going to provide polynomial time algorithms that find construction schedules for graphs that belong to specific graph classes. We will start from basic graphs such as the line and the star whose schedules we will use as sub-schedules later on. Note here that for the purposes of clear and easy to understand descriptions, we will sometimes provide a straight construction of a graph  $G$  instead of a construction schedule. Both are equivalent.

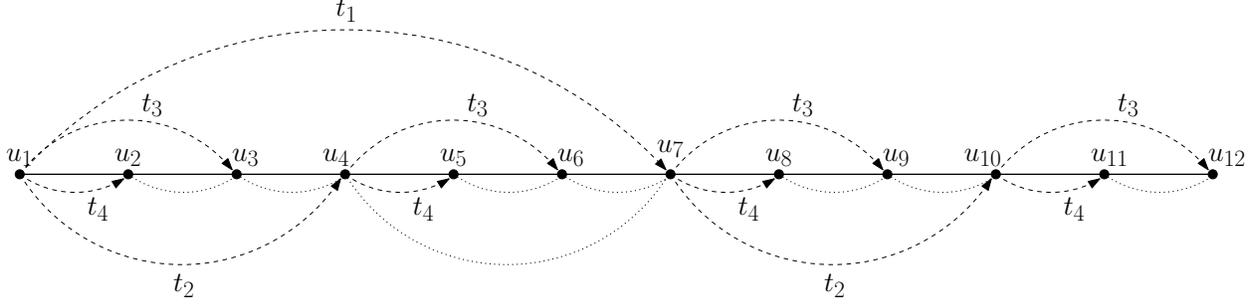


Figure 1: In this figure, the dashed arrows represent node generations in slots  $t_i = i$ . The dotted lines represent excess edges.

### 3.1 Line

We here provide a polynomial-time algorithm, called *line construction schedule*, that computes a construction schedule for a line graph  $G$  with  $\lceil \log n \rceil$  slots. Let  $G = (V, E)$  be the line graph of size  $n$  that we need to construct, where  $V = \{u_1, u_2, \dots, u_n\}$ , and each  $u_i \in V$ ,  $2 \leq i \leq n - 1$ , is connected with  $u_{i-1}$  and  $u_{i+1}$ . The general strategy that we need to follow in order to construct a line in exactly  $\lceil \log n \rceil$  slots is that in each slot  $t$ , all nodes of  $G_t$  must generate new nodes, except possibly for the last slot. Therefore, the problem is to assign to each node a set of nodes of  $V$  that it needs to generate throughout all time slots, while guaranteeing that the final graph will have a spanning line as a sub-graph. In particular, each node  $u_i \in G_t$  generates the node  $u_j = u_{i+\lceil n/2^t \rceil}$  in slot  $t$ , if  $u_j \in V$ , and activates edge  $u_i u_j$ . In addition, if  $u_{(i+\lceil n/2^{(t-1)} \rceil)} \in V_t$ , then activates the edge  $u_j u_{(i+\lceil n/2^{(t-1)} \rceil)}$ . See Figure 1 for an example and Algorithm 2 for the pseudo code. In essence, this means that the *initiator*  $u_1$  will first generate the middle node of the line, then, in the next slot, the middle between itself and the node that it ( $u_0$ ) generated in the previous slot, and so on. This procedure is repeated until it generates its neighbor on the line graph  $G$ . In a similar way, in each slot  $t$ , all nodes construct the middle node between themselves and the next node of the line that was generated in the previous slot  $t - 1$ . In case that such a node does not exist, they generate the middle node between themselves and the end of the line (node  $u_n$  that has not been generated yet). Finally, each node  $u_j$  which is generated by  $u_i$  activates an additional edge with the next node of the line that  $u_i$  has edge with (if any).

**Lemma 6.** *The line construction schedule algorithm computes in polynomial time a construction schedule of a line graph of size  $n$  with  $\lceil \log n \rceil$  slots and with  $O(n)$  excess edges.*

*Proof.* It is easy to see by the description and by Figure 1 above that the graph constructed has a line subgraph of size  $n$ . The construction schedule has  $\lceil \log n \rceil$  slots by design. For the excess edges, consider that in the whole construction schedule, for every node generation there are at most two edge activations. Since there are  $n - 1$  nodes generated in total, there are  $2(n - 1)$  total edge activations. Therefore, the excess edges are at most  $2(n - 1) - (n - 1) = O(n)$ .  $\square$

### 3.2 Star

**Lemma 7.** *There is a polynomial time algorithm that computes in polynomial time a construction schedule of a star graph of size  $n$  with  $\lceil \log n \rceil$  slots and with  $O(n)$  excess edges.*

*Proof.* Let  $u$  be the initiator and  $n$  the size of the star graph. Let  $G_t = (V_t, E_t)$  be the generated graph at slot  $t$ . Then, in each slot the graph  $G_{t+1}$  is obtained from  $G_t$  as follows: Assume that

---

**Algorithm 2** Line construction schedule

---

**Input:** A line graph  $G = (V, E)$  with  $n$  nodes.

**Output:** A valid construction schedule for  $G$ .

```
1:  $V = u_1$ 
2: for  $k = 1, 2, \dots, \lceil \log n \rceil$  do
3:    $\mathcal{S}_k = \emptyset$ 
4:    $V_k = \emptyset$ 
5:   for each node  $u_i \in V_f$  do
6:      $\mu(t) = i + \lceil n/2^t \rceil$ 
7:     if  $u_{\mu(t)} \in V$  then
8:        $\mathcal{S}_k = \mathcal{S}_k \cup \{(u_i, u_{\mu(t)}), \{u_i u_{\mu(t)}, (u_{\mu(t)} u_{\mu(t-1)} : u_{\mu(t-1)} \in V)\}\}$ 
9:        $V_k \leftarrow V_k \cup u_{\mu(t)}$ 
10:   $V \leftarrow V \cup V_k$ 
11: return  $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{\lceil \log n \rceil})$ 
```

---

$G_t$  has a star subgraph with center the node  $u$ . Then, each node which is generated in slot  $t + 1$ , is in distance 2 from  $u$ , thus it activates an edge with  $u$ . Observe that  $G_{t+1}$  still has a star as a subgraph, with  $u$  as its center. This process is repeated until slot  $\lceil \log n \rceil$ . Finally, in the last slot  $\lceil \log n \rceil$ ,  $n - 2^{\lceil \log n \rceil}$  arbitrary nodes generate new nodes, while the rest of them remain idle.

For every node generation, there are at most two edge activations. Since there are  $n - 1$  nodes generated in total, there are  $2(n - 1)$  total edge activations. Therefore, the excess edges are at most  $2(n - 1) - (n - 1) = O(n)$ .  $\square$

### 3.3 Trees

The strategy that we follow to construct any tree in logarithmic time is based on ideas of our line and star construction schedules. For the purposes of an easy and clear to understand description, we are going to describe the construction instead of using a construction schedule. The *Tree algorithm* works as follows: Let  $G = (V, E)$  be any tree of size  $n$  and diameter  $diam$ . We first assume that the initiator corresponds to the node  $u$  which is at distance  $\lceil diam/2 \rceil$  from an arbitrary endpoint of the longest path of the tree. We hereafter consider the graph  $G_u$  to be  $G$  rooted at  $u$ . Let  $L_i^w$  denote all nodes in distance  $i$  from node  $w$  on its sub-tree, and  $d_w$  be the depth of its sub-tree. We execute a combination of the line and star construction schedules in parallel as follows:

Each node can be in one of the states *center* and *leaf*, and initially it is in state *leaf*, except from the initiator which starts from the former one. In the *center* state the algorithm proceeds in phases. Let  $p_w$  denote the phase of node  $w$  (initially  $p_w = 1$ ). In each phase,  $w$  executes the star algorithm with the nodes of  $L_i^w$ , where  $i = \lceil d_w/2^{p_w} \rceil$ . The newly generated nodes that are initially in state *leaf* execute the star algorithm with  $w$  as the center of the star. When the star algorithm is completed (i.e., all nodes of  $L_i^w$  have been generated), all nodes of the star enter to state *center*, and  $w$  increases its phase by one.

During the execution of the star algorithm, the nodes of  $L_i^w$  activate some additional edges with the nodes in the next (i.e., lower) level of the tree that  $w$  has edges with (if any). Let  $L$  and  $L'$  be two such sets of nodes, where  $L'$  is in a higher level of the tree. Each node  $u \in L'$  will eventually activate all edges with the nodes of  $L$  that belong to its sub-tree (call this set of nodes  $L_u$ ). To achieve this, let  $l_u$  be the birth path of  $u$ . Then, each node  $w \in l_u$  activates all edges with  $L_u$ . Observe that this is possible as  $w$  is the center of a star with leaves the nodes of  $L$ .

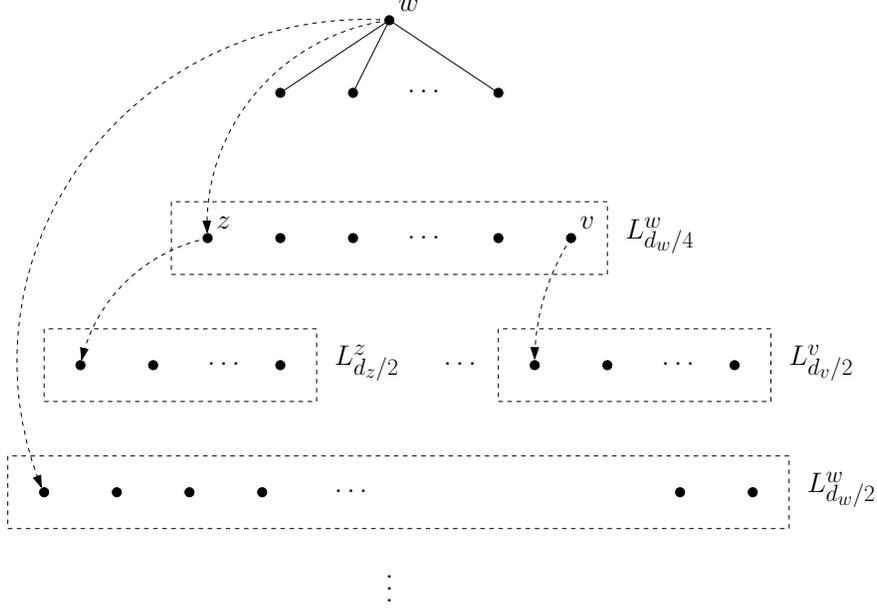


Figure 2: In this figure, the dashed arrows, starting from a node  $u$ , represent the construction of a star with center the node  $u$ , and leaves all nodes in the corresponding dashed rectangles. In the above example,

This ends the description of the Tree algorithm. See Figure 2 for an example.

**Lemma 8.** *The Tree algorithm constructs a given tree  $G$  with  $O(\log^2 n)$  slots, and with  $O(n \log^2 n)$  excess edges.*

*Proof.* For simplicity, assume that the construction of the tree proceeds in phases, where each phase starts with the construction of some level or levels of the tree, and it is completed when all nodes of these levels have been generated.

In each phase  $i$ , the number of levels that we construct is  $2^i$ , except possibly for the last round. Therefore, the number of phases is bounded by  $O(\log n)$ . During each phase, all nodes of the tree that have been generated execute the *star* algorithm with some nodes that belong to their sub-trees, and run in parallel as these executions are independent between each other. The number of rounds in each phase can be bounded by  $O(\log n)$ . Therefore, the total execution time of the algorithm is  $O(\log^2 n)$  number of rounds.  $\square$

We will now use a different algorithmic strategy in order to minimize the excess edges. Note that our previous strategy included splitting the tree graph into levels and creating the middle levels in each slot. This technique requires a lot of excess edges because nodes are generated by quite distant nodes in the final graph. Therefore, we propose the following algorithm which improves the number of excess edges to  $O(n)$ .

Let  $G_f = (V_f, E_f)$  be the target tree graph. Our algorithm is based on applying a decomposition strategy on the final graph  $G_f$  where nodes and edges are removed in phases, until graph  $G_f$  only has a single node. Afterwards, we will show that each phase can be reversed using a construction schedule. Finally, we will show that we can always combine the construction schedules of each phase into a single schedule. The decomposition consists of two phases that are interchanging between themselves.

### Efficient Schedule for Trees Algorithm

**Line-cut phase:** For each line subgraph  $G' = (V', E')$ , where  $V'$  contains nodes  $u_1, u_2, \dots, u_k$ , for  $k \leq n$ , and  $u_2, u_3, \dots, u_{k-1}$  have degree  $deg = 2$  in  $G_f$  and  $u_1, u_k$  have degree ( $deg = 1 \vee deg \geq 3$ ) in  $G_f$ , we activate an edge between the two endpoints  $u_1, u_k$  and remove every node  $u_2, u_3, \dots, u_{k-1}$  along with their incident edges. An example of this procedure is shown in Figure 3. If  $G_f$  contains a single node, terminate; otherwise proceed to leaf-cut phase.

**Leaf-cut phase:** In this phase, every leaf node of the graph is removed along with its incident edges. An example of this procedure is shown in Figure 4. If  $G_f$  contains a single node, terminate; otherwise proceed to line-cut phase.

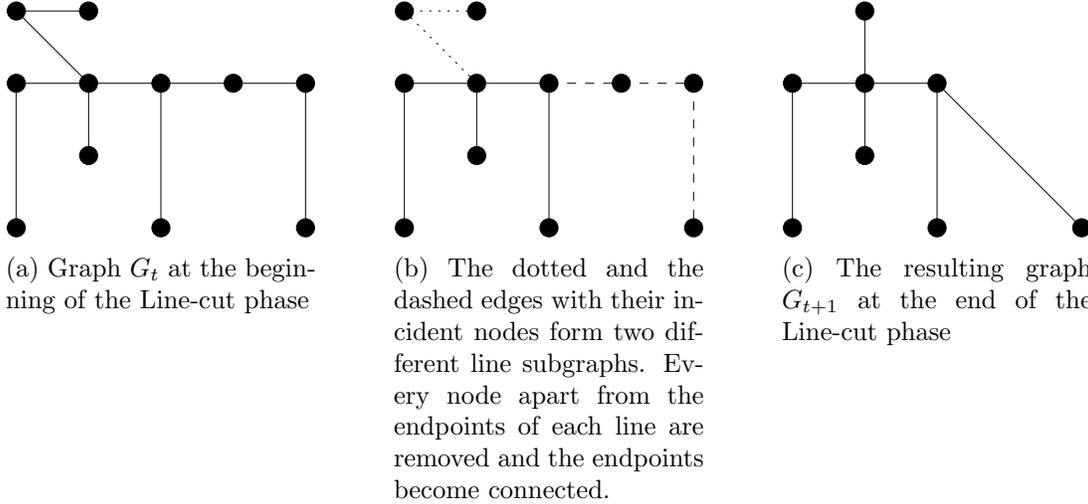


Figure 3: An example of a Line-cut phase

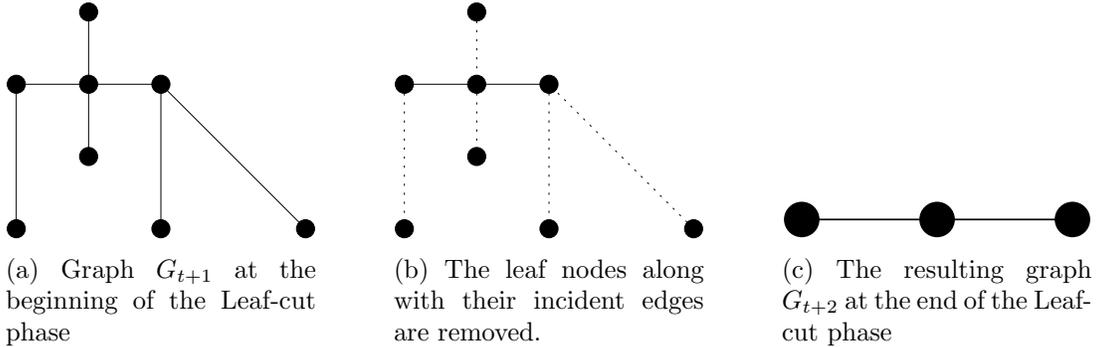


Figure 4: An example of a Leaf-cut phase

**Lemma 9.** *Given any tree graph  $G_f = (V_f, E_f)$ , the efficient schedule for trees algorithm deconstructs  $G_f$  into a single node using  $O(\log n)$  phases.*

*Proof.* Consider the graph  $G_i = (V_i, E_i)$  after the execution of an arbitrary line-cut phase  $p$ . The line cut phase removes every node that has 2 neighbors in the graph, and in phase  $p + 1$  which is a leaf-cut phase, the graph consists of leaf nodes  $u \in S_u$  and internal nodes  $v \in S_v$  with  $deg > 2$ . Therefore,  $|S_u| > |S_v|$  and since  $|S_u| + |S_v| = |V_i|$ , we can surmise that  $|S_u| > |V_i|/2$  and any leaf-cut phase cuts the size of the current graph in half. Thus, there can be at most  $\log n$  leaf-cut phases and  $\log n$  line cut phases before the graph only has a single node.  $\square$

**Lemma 10.** *Every phase can be reversed using a construction schedule with  $O(\log n)$  slots.*

*Proof.* First, let us consider the line-cut phase. In this phase, every starting subgraph  $G'$  is a line subgraph with nodes  $u_1, u_2, \dots, u_k$ , where  $u_1, u_k$  are the endpoints of the line. At the end of the phase, every subgraph has two connected nodes  $u_1, u_k$ . The reversed process works as follows: we use node  $u_1$  as the initiator and we execute our line construction schedule in order to generate nodes  $u_2, u_3, \dots, u_{k-1}$ . In addition to that, every time a node is generated, it also activates an edge with node  $u_k$ . After the construction is finished, nodes deactivate the edges not belonging to the original line subgraph  $G'$ . This construction schedule requires  $\log k \leq \log n$  slots.

Now let us consider the leaf-cut phase. In this phase, every starting subgraph  $G'$  is a leaf  $u'$  and its neighbor  $v'$ . If any node  $v'$  belongs to multiple subgraphs  $G'$ , which means that  $v'$  is a neighbor to multiple leaves, we combine all these subgraphs together into a single subgraph  $G''$  which is a star subgraph with center the node  $v'$  and leaf nodes  $u'_1, u'_2, \dots, u'_k$ . At the end of the phase, every subgraph  $G''$  only contains the node  $v'$ . The reversed process works as follows: we use node  $v'$  as the initiator and execute our star construction schedule to generate nodes  $u'_1, u'_2, \dots, u'_k$ . After the construction is finished, nodes deactivate the edges not belonging to the original star subgraph  $G''$ . This construction schedule requires  $\log k \leq \log n$  slots.  $\square$

**Lemma 11.** *The construction schedules of each phase can be combined into a single construction schedule with  $O(\log^2 n)$  slots and with  $O(n)$  excess edges.*

*Proof.* The construction schedules can be straightly combined into a single one, since every sub-schedule  $\sigma_i$  uses only a single node as an initiator  $u$ , which is always available (i.e.,  $u$  was generated by some previous  $\sigma_j$ ). Since we have  $O(\log n)$  schedules and every schedule has  $O(\log n)$  slots, the combined construction schedule needs  $O(\log^2 n)$  slots. By Lemma 10, in every slot there are  $O(k)$  excess edges, where  $k$  is the number of nodes generated in that slot. Since the final graph has  $n$  nodes, the excess edges activated throughout the whole schedule are  $O(n)$ .  $\square$

**Theorem 2.** *The efficient schedule for trees algorithm is a polynomial time algorithm, that can find a schedule that computes a construction schedule for any given tree graph  $G_f = (V_f, E_f)$ , starting from a single node, with  $O(\log^2 n)$  slots and with  $O(n)$  excess edges.*

*Proof.* Follows from Lemmas 10 and 11.  $\square$

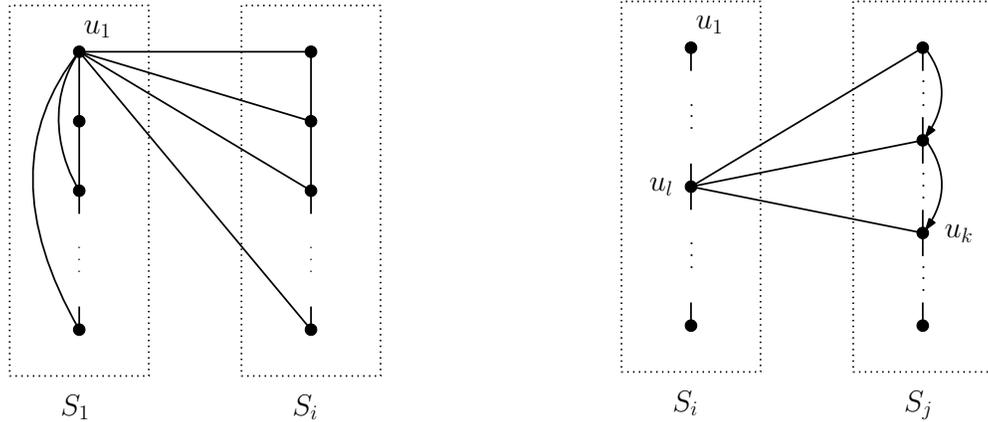
### 3.4 Planar Graphs

Our goal is to provide an algorithm that computes a construction schedule for any given planar graph starting from a single node. Let  $G_f = (V_f, E_f)$  be the target planar graph. Starting from another graph  $G = (V, E)$ , where set  $V$  contains a single node and  $E = \emptyset$ , our schedule transforms  $G$  so that  $V = V_f$  and  $E = E_f$ . When a node  $u$  is generated in graph  $G$  it is matched with a node  $u_f \in V_f$ . Let  $u, v \in G$  be two nodes that are matched with  $u_f, v_f \in G_f$ , respectively. If  $(u_f, v_f) \in E_f$ , then  $(u, v) \in E$ . First, we use any algorithm that can give us a 5-coloring of our planar graph  $G_f$  in linear time and split the nodes into five sets  $S_i$ ,  $1 \leq i \leq 5$ . Our schedule will have five sub-schedules, and in every sub-schedule  $i$  we generate every node of set  $S_i$ . In every sub-schedule, we use a modified version of our line construction schedule. We call this algorithm, *planar graph*.

**Preprocessing:** We use any algorithm that can give us a 5-coloring of our planar graph  $G_f$  in linear time. We create sets  $S_i \subseteq V_f$ , where  $1 \leq i \leq 5$ , and every set  $S_i$  contains nodes with the same color. W.l.o.g. we can assume that  $|S_1| \geq |S_2| \geq |S_3| \geq |S_4| \geq |S_5|$ . Note here that in the given graph, every set  $S_i$  is an independent set.

**Planar Graph Construction Schedule:** In every sub-schedule  $i$ , using node  $u_1$  as the initiator, we use the line construction schedule that generates  $|S_i|$  nodes with the following addition: whenever any node  $u_k$  is generated, where  $k = 1, 2, \dots, |S_i|$ , it also activates an edge with  $u_1$ . We call this new construction schedule *Anchor Line* (see Figure 5a). Additionally, we uniquely match every node  $u_k$  with an arbitrary node  $w_k \in S_i$  that has not been matched yet with any other node. For every edge  $(w_l, w_k) \in E_f$  we add the edge  $(u_l, u_k)$  in the construction schedule, where  $w_l$  and  $w_k$  are matched with  $u_l$  and  $u_k$ , respectively. Finally, for every node  $u_j \in B_{u_k}$ , we add the edges  $(u_j, u_l)$  in the construction schedule (see Figure 5b).

Node  $u_1$  belongs to set  $S_1$  and acts as the initiator of all  $S_i$ ,  $1 \leq i \leq 5$ . Once all nodes of  $S_i$  have been generated,  $u_1$  starts the construction of  $S_{i+1}$ .



(a) *Anchor Line*. All nodes of each set  $S_i$  form a line and are additionally connected with the initiator  $u_1$ .

(b) Arrows represent node generations. For each pair of connected nodes in the target graph  $G_f$ , the nodes  $u_l \in S_i$  and  $u_k \in S_j$  which are mapped onto them activate an edge between them,  $u_l$  activates edges with all nodes on the birth path of  $u_k$ .

Figure 5: Planar graphs construction

**Lemma 12.** *The planar graph construction schedule correctly transforms the starting graph  $G = (V, E)$ , where  $|V| = 1$  and  $E = \emptyset$ , into the target planar graph  $G_f = (V_f, E_f)$ , so that  $V = V_f$  and  $E = E_f$ .*

*Proof.* Based on the description of the schedule, it is easy to see why  $V = V_f$ , since we break  $V_f$  into our five sets  $S_i$  and we generate each set in a different phase  $i$ . This is always possible no matter the graph  $G_f$ , since every set  $S_i$  is an independent set.

We will now prove that  $E = E_f$ . Consider any node  $u_i$  that is generated in sub-schedule  $i$ , and any node  $u_j$  generated in phase  $j$ , where  $j < i$ . Let us assume that node  $u_i$  is matched with node  $w_i \in S_i$  and node  $u_j$  is matched with node  $w_j \in S_j$ . If  $(w_i, w_j) \in E_f$ , we must activate  $(u_i, u_j)$  once  $u_i$  is generated. we know that this is always possible, because the construction schedule activates an edge between any node of the birth path of  $u_i$  with  $u_j$ .  $\square$

**Lemma 13.** *The planar graph construction schedule has  $O(\log n)$  time slots and  $O(n \log n)$  excess edges.*

*Proof.* Let  $n_i$  be the size of the independent set  $S_i$ . Then, the sub-schedule that constructs  $S_i$  requires the same slots as our line construction schedule, which is  $\lceil \log n_i \rceil$  slots. Combining the

five sub-schedules would require:

$$\sum_{i=1}^5 \log n_i = \log \prod_{i=1}^5 n_i < 5 \log n = O(\log n) \quad (1)$$

Let us consider the excess edges activated in every sub-schedule. The number of excess edges activated are those of the *Anchor Line* schedule and the excess edges for the birth path of each node. The excess edges of the *Anchor Line* schedule are  $O(n)$ . We also know that the birth path of each node  $u$  includes at most  $|B_u| = O(\log n)$  nodes. Since we have a planar graph we know that there are at most  $3n$  edges in graph  $G_f$ . For every edge  $(u, v)$  in the target graph, we would need to add  $|B_u|$  additional edges. Therefore, no matter the structure of the  $3n$  edges, the schedule would activate  $3nO(\log n) = O(n \log n)$  excess edges.  $\square$

**Theorem 3.** *The planar graph algorithm computes in polynomial time a construction schedule for any given planar graph  $G_f = (V_f, E_f)$  with  $O(\log n)$  slots and with  $O(n \log n)$  excess edges.*

*Proof.* Follows from Lemmas 12 and 13.  $\square$

**Corollary 1.** *The planar graph algorithm can be extended to find a construction schedule in polynomial time for any graph  $G = (V, E)$  with  $O(k \log n)$  slots and with  $O(l \log n)$  excess edges, where  $k$  is a valid  $k$ -coloring for graph  $G$  given as an input, and  $l = |E|$ .*

*Proof.* The extension of our *planar graph* algorithm is possible by running  $k$  sub-schedules, where each one creates a set  $S_i \subseteq V$ ,  $i = 1, 2, \dots, k$ , and each  $S_i$  contains nodes with the same color.  $\square$

**Corollary 2.** *The planar graph algorithm can be extended to find a construction schedule in polynomial time for any graph  $G = (V, E)$  with  $(\Delta + 1)(\log n)$  slots and with  $O(l \log n)$  excess edges, where  $\Delta$  is the maximum degree of  $G$  and  $l = |E|$ .*

*Proof.* We can use a greedy polynomial time algorithm to find a  $\Delta + 1$  coloring and extend the *planar graph* algorithm.  $\square$

## 4 Construction Schedules with zero-waste

In this section, we study graphs that can be constructed with  $\ell = 0$  excess edges. The goal is to find the shortest construction schedule for any given graph  $G$ , provided that there will be no edge deletions at all. This means that every edge generated by a construction schedule  $\sigma$ , must belong to  $G$ . We begin by checking whether a graph  $G$  has any valid construction schedule with  $\ell = 0$  excess edges. Due to the following observation, it suffices to check for construction schedules with  $n - 1$  slots.

**Observation 5.** *Graph  $G$  has no valid construction schedule with zero excess edges and  $k$  slots if there is no valid construction schedule with zero excess edges and  $k' \geq k$  slots.*

**Definition 2.** *A cop-win order of a graph  $G$  is an ordering  $v_1, v_2, \dots, v_n$  of  $V(G)$  such that  $v_i$  is dominated in the subgraph induced by  $v_i, \dots, v_n$ , for  $1 \leq i \leq n$ . A node  $v$  is said to be dominated by another node  $w$  when  $N[v] \subseteq N[w]$ . A graph that admits a cop-win ordering, is a cop-win graph.*

**Definition 3.** *Let  $G = (V, E)$  be any graph. A node  $v \in V$  is a candidate node for being the last node in a construction schedule  $\sigma$  for  $G$  if there exists a node  $w \in V \setminus \{v\}$  such that  $N[v] \subseteq N[w]$ . In this case  $w$  is a candidate parent of  $v$ . Furthermore the set of candidate nodes in  $G$  is denoted by  $S_G = \{v \in V : N[v] \subseteq N[w] \text{ for some } w \in V \setminus \{v\}\}$ . See Figure 6.*

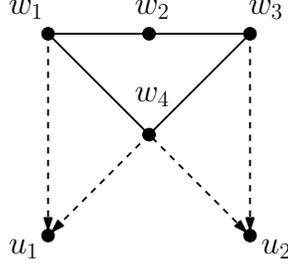


Figure 6: Node  $u_1$  and  $u_2$  are called candidate nodes. The arrows represent all possible potential node generations in the previous time slot. In particular, nodes  $w_1$  and  $w_4$  are candidate parents of  $u_1$ , while  $w_3$  and  $w_4$  are candidate parents of  $u_2$ .

---

**Algorithm 3** Recognition of cop-win graphs

---

**Input:** A graph  $G = (V, E)$  with  $n$  nodes.

**Output:** A valid construction schedule for  $G$ , or the announcement that  $G$  is not a cop-win graph.

- 1: **for**  $k = n - 1$  **downto** 1 **do**
  - 2:      $\mathcal{S}_k = \emptyset$
  - 3:     **for** every node  $v \in V$  **do**
  - 4:         **if**  $(N[v] \subseteq N[u], \text{ for some node } u \in V \setminus \{v\}) \wedge (\mathcal{S}_k = \emptyset)$  **then**  $\{v \text{ is a new candidate node}\}$
  - 5:          $\mathcal{S}_k \leftarrow \{(u, v, \{vw : w \in N(v)\})\}$
  - 6:          $V \leftarrow V \setminus \{v\}$
  - 7:     **if**  $\mathcal{S}_k = \emptyset$  **then**
  - 8:         **return** “NO”
  - 9: **return**  $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{n-1})$
- 

**Lemma 14.** *Graph  $G = (V, E)$  has a construction schedule with  $n - 1$  slots and  $\ell = 0$  excess edges, if and only if  $G$  is a cop-win graph.*

*Proof.* By definition of the model, whenever a node  $u$  is generated from node  $w$  in a slot  $k$ , node  $u$  can activate edges only with nodes in  $N[w]$  which means that  $N[u] \subseteq N[w]$  and since  $\ell = 0$ , this property stays true in  $G_{k+1}$ . Therefore, any node  $u$  generated in slot  $k$ , is dominated by its parent in graph  $G_{k+1}$ .  $\square$

Cop-win graphs have been studied extensively in the past and very efficient recognition algorithms have been established prior to this work. For the benefit of the readers, we provide our own recognition algorithm since we are going to use similar strategies in other algorithms. The algorithm can decide whether a graph has a cop-win ordering, and therefore, whether it can be constructed with a schedule with  $n - 1$  slots and  $\ell = 0$  excess edges. Given the graph  $G = (V, E)$ , the algorithm finds all candidate nodes that could have been generated in the last slot and arbitrarily removes one of them and its incident edges. The removed node is put in the last empty slot of the schedule. The algorithm repeats the above process until there is only a single node left. If that is the case, the algorithm decides that the graph is cop-win and produces a construction schedule. If the algorithm cannot find any potential nodes for removal, it decides that the graph is not cop-win.

**Lemma 15.** *Let  $v \in S_G$ . Then  $G$  has a cop-win ordering if and only if  $G - v$  has a cop-win ordering.*

*Proof.* Let  $c$  be a cop-win ordering of  $G - v$ . Then, generating node  $v$  at the end of  $c$  trivially results in a cop-win ordering of  $G$ .

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

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

**Theorem 4.** *The recognition of cop-win graphs can decide in polynomial time, whether a given graph  $G = (V, E)$  is cop-win, and if so, it also produces a construction schedule with  $n - 1$  slots and  $\ell = 0$  excess edges.*

*Proof.* First note that we can find the candidate nodes in polynomial time, and thus the algorithm terminates in polynomial time. This is because the algorithm removes one candidate node  $u$  in each loop, which based on Lemma 15, graph  $G - u$  is cop-win if only if  $G$  was cop-win. This means that the algorithm terminates with a construction schedule for  $G$  if and only if  $G$  is cop-win.  $\square$

**Lemma 16.** *There is a modified version of the recognition of cop-win graphs algorithm that computes in polynomial time a construction schedule for any graph  $G$  with  $n - 1$  slots and  $\ell$  excess edges, where  $\ell$  is a constant, if and only if such a schedule exists.*

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

Our next goal is to decide whether a graph  $G = (V, E)$  with  $n = 2^\delta$  nodes has a valid construction schedule  $\sigma$  with  $\log n$  slots and  $\ell = 0$  excess edges. The fast cop-win algorithm first finds all candidate nodes  $S_G$  that can be generated in the last slot. It then tries to find a valid subset  $L \subseteq S_G$  of candidates that satisfies all of the following:

- $|L| = n/2$ .

- $L$  must be an independent set.
- There is a perfect matching between the candidate nodes in  $L$  and their candidate parents.

The algorithm finds such a set by creating a 2-SAT formula  $\phi$  whose solution is a valid set  $L$ . If the algorithm finds such a set  $L$ , it adds the nodes in  $L$  to the last slot of the construction schedule. It then removes nodes in  $L$  from graph  $G$  along with their incident edges. The above process is then repeated to find the next slots. If at any point, graph  $G$  has a single node, the algorithm terminates and outputs the construction schedule. If at any point, the algorithm cannot find a valid set  $L$ , it outputs “no” since there is no valid construction schedule for  $G$ . This algorithm is shown in Algorithm 4.

**Lemma 17.** *Consider any graph  $G = (V, E)$ .  $G$  has a valid construction schedule with  $\log n$  slots and  $\ell = 0$  excess edges if and only if there exists a perfect matching  $M$  that contains a valid candidate node set  $L$ , where  $L$  has exactly one node for each edge of the perfect matching  $M$ .  $L$  is a valid candidate node set if  $|L| = n/2$ , and is an independent set.*

*Proof.* Let us assume that graph  $G$  has a valid construction schedule. Then in the last slot, there are  $n/2$  nodes, called parents, that generated  $n/2$  other nodes, called children. Therefore, such a perfect matching  $M$  always exists where set  $L$  contains the children. Similarly, if there is no perfect matching in  $G$ , then there is no valid construction schedule due to fact that in any construction schedule for  $G$ , the last slot creates a perfect matching in  $G$ .  $\square$

**Lemma 18.** *The 2-SAT formula  $\phi$ , generated by the fast cop-win algorithm, has a solution if and only if there is an independent set  $|V_2| = n/2$ , where  $V_2$  is a valid set of candidate nodes in graph  $G = (V, E)$ .*

*Proof.* Let us assume that graph  $G$  has a valid construction schedule. Based on Lemma 17, there are  $n/2$  parents and  $n/2$  children in  $G$ . Therefore, there has to be a set  $V_2$ , where  $|V_2| = n/2$  and  $V_2$  is an independent set such that there is another set  $V_1$ , where  $|V_1| = n/2$  and  $V_1 \cap V_2 = \emptyset$ . Any perfect matching  $M \in G$  includes edges  $u_i v_i \in M$ , where  $u_i \in V_1$  and  $v_i \in V_2$  because  $V_2$  is an independent set.

The solution to the 2-SAT formula  $\phi$  we are going to create is a valid set  $V_2$  as stated above. Consider an arbitrary edge  $u_i v_i$  from the perfect matching  $M$ . The algorithm creates a variable  $x_i$  for each  $u_i v_i$ . The truthful assignment of  $x_i$  means that we pick  $v_i$  for  $V_2$  and the negative assignment means that we pick  $u_i$  for  $V_2$ . Since  $|V_2| = n/2$ , then for every edge  $u_i v_i \in M$ , at least one of  $u_i, v_i$  is a candidate node, because otherwise some other edge  $u_j v_j \in M$  would need to have 2 candidates nodes at its endpoints and include them both in  $V_2$ , which is impossible. Thus, graph  $G$  would have no valid construction schedule.

If  $v_i$  is a candidate node and  $u_i$  is not, then  $v_i \in V_2$ , and we add clause  $(x_i)$  to  $\phi$ . If  $u_i$  is a candidate node and  $v_i$  is not, then  $u_i \in V_2$ , in which case we add clause  $(\neg x_i)$  to  $\phi$ . If both  $u_i$  and  $v_i$  are candidate nodes, either one could be in  $V_2$  as long as  $V_2$  is an independent set.

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

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

---

**Algorithm 4** Fast cop-win

---

**Input:** A graph  $G = (V, E)$  with  $n = 2^\delta$  nodes.

**Output:** An construction schedule with  $k = \log n$  slots and  $\ell = 0$  excess edges for  $G$ .

```
1: for  $k = \log n$  downto 1 do
2:    $\mathcal{S}_k = \emptyset$ ;  $\phi = \emptyset$ 
3:   Find a perfect matching  $M = \{u_i v_i : 1 \leq i \leq n/2\}$  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  $(N[u_i] \subseteq N[w], \text{ for some node } w \in V \setminus \{u_i\}) \wedge (N[v_i] \not\subseteq N[x], \text{ for any node } x \in V \setminus \{v_i\})$ 
10:    then  $\{u_i \text{ is a candidate node and } v_i \text{ is not.}\}$ 
11:     $\phi \leftarrow \phi \wedge (\neg x_i)$ 
12:   else if  $(N[u_i] \not\subseteq N[w] \text{ for any node } w \in V \setminus \{u_i\}) \wedge (N[v_i] \subseteq N[x], \text{ for some node } x \in V \setminus \{v_i\})$ 
13:   then  $\{u_i \text{ is a not candidate and } v_i \text{ is a candidate}\}$ 
14:    $\phi \leftarrow \phi \wedge (x_i)$ 
15:   else if  $(N[u_i] \not\subseteq N[w], \text{ for some node } w \in V \setminus \{u_i\}) \wedge (N[v_i] \not\subseteq N[x], \text{ for some node } x \in V \setminus \{v_i\})$ 
16:   then  $\{u_i \text{ is not a candidate and } v_i \text{ is not a candidate}\}$ 
17:   return "NO"
18:   for every edge  $u_i u_j \in E \setminus M$  do
19:      $\phi \leftarrow \phi \wedge (x_i \vee x_j)$ 
20:   for every edge  $v_i v_j \in E \setminus M$  do
21:      $\phi \leftarrow \phi \wedge (\neg x_i \vee \neg x_j)$ 
22:   for every edge  $u_i v_j \in E \setminus M$  do
23:      $\phi \leftarrow \phi \wedge (x_i \vee \neg x_j)$ 
24:   if  $\phi$  is satisfiable then
25:     Let  $\tau$  be a satisfying truth assignment for  $\phi$ 
26:     for  $i = 1, 2, \dots, n/2$  do
27:       if  $x_i = \text{true}$  in  $\tau$  then
28:          $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup (u_i, v_i, \{v_i w : w \in N(v_i)\})$ 
29:          $V \leftarrow V \setminus \{v_i\}$ 
30:          $E \leftarrow E \setminus \{v_i w : w \in N(v_i)\}$ 
31:       else  $\{x_i = \text{false} \text{ in } \tau\}$ 
32:          $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup (v_i, u_i, \{u_i w : w \in N(u_i)\})$ 
33:          $V \leftarrow V \setminus \{u_i\}$ 
34:          $E \leftarrow E \setminus \{u_i w : w \in N(u_i)\}$ 
35:     else  $\{\phi \text{ is not satisfiable}\}$ 
36:     return "NO"
37:   return  $\sigma = (\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k)$ 
```

---

**Lemma 19.** *Consider any graph  $G = (V, E)$ . If  $G$  has a valid construction schedule with  $\log n$  slots and  $\ell = 0$  excess edges, then any arbitrary perfect matching contains a valid candidate set  $|L| = n/2$ , where  $L$  has exactly one node for each edge of the perfect matching.*

*Proof.* By Lemma 18, any perfect matching  $M$  contains edges  $uv$ , such that there exists a valid candidate set  $V_2$  that contains one node exactly for each edge  $uv \in M$ . Thus, if graph  $G$  has a

valid construction schedule, the solution to the 2-SAT formula corresponds to a valid candidate set  $V_2$ .  $\square$

**Theorem 5.** *The fast cop-win algorithm computes in polynomial time a construction schedule  $\sigma$  for any graph  $G = (V, E)$ , where  $|V| = 2^\delta$ , with  $\log n$  slots and  $\ell = 0$  excess edges, if and only if such a  $\sigma$  exists for  $G$ .*

*Proof.* Suppose that  $G = (V, E)$  has a construction schedule  $\sigma$  with  $\log n$  slots and  $\ell = 0$  excess edges. By Lemmas 18 and 19 we know that our *fast cop-win* algorithm finds a set  $L$  for the last slot of a schedule  $\sigma''$  but this might be a different set from the last slot contained in  $\sigma$ . Therefore, for our proof to be complete, we need to show that if  $G$  has a construction schedule  $\sigma$  with  $\log n$  slots and  $\ell = 0$  excess edges, for any  $L$  it holds that  $(G - L)$  has a construction schedule  $\sigma'$  with  $\log n - 1$  slots and  $\ell = 0$  excess edges.

Assume that  $\sigma$  has in the last slot  $\mathcal{S}_k$  a set of nodes  $V_1$  generating another set of nodes  $V_2$ , 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 node in  $V_s$ . Since  $u' \notin V_2$  and  $u'$  has no edge with any node 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 nodes in any construction schedule and still maintain a valid construction schedule. Therefore, for  $L = V_2'$ , the graph  $(G - L)$  has a construction schedule  $\sigma'$  with  $\log n - 1$  slots and  $\ell = 0$  excess edges.

Assume now that  $V_2' \cap V_2 = V_s$ , where  $|V_s| = x \geq 0$ . Then,  $V_2' = V_s \cup u'_1 \cup u'_2, \dots \cup u'_y$  and  $V_2 = V_s \cup u_1 \cup u_2, \dots \cup u_y$ , where  $y = n/2 - x$ . As argued above, nodes  $u'_1, u'_2, \dots, u'_y$  can be candidate parents only to nodes  $u_1, u_2, \dots, u_y$ , 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  $j = 1, 2, \dots, y$ . Thus these nodes can be swapped in the construction schedule and still maintain a valid construction schedule. Therefore for any arbitrary  $L = V_2'$ , the graph  $(G - L)$  has a construction schedule  $\sigma'$  with  $\log n - 1$  slots and  $\ell = 0$  excess edges.  $\square$

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

**Definition 4.** *Given any graph  $G$ , find a valid construction schedule with  $\kappa$  slots (minimum) and  $\ell = 0$  excess edges. We call this problem zero-waste construction schedule.*

**Theorem 6.** *The decision version of the zero-waste construction schedule 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 construction schedule  $\sigma$ , together with an isomorphism between the graph constructed by  $\sigma$  and the target graph  $G$ .

To show NP-hardness, we provide a reduction from the coloring problem. Given an arbitrary graph  $G = (V, E)$  with  $n$  nodes, we construct graph  $G' = (V', E')$  as follows: Let  $G_1 = (V_1, E_1)$  be an isomorphic copy of  $G$ , and let  $G_2$  be a clique of  $n$  nodes.  $G'$  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 node of  $G_2$  is a universal node in  $G'$  (i.e., a node which is connected with every other node 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 construction schedule for  $G'$ . We will show that  $\kappa(G') = \chi(G) + n$ .

Let  $\sigma$  be an optimal construction schedule for  $G'$ , which uses  $\kappa(G')$  time slots. As every node  $v \in V_2$  is a universal node in  $G'$ ,  $v$  cannot coexist with any other node of  $G'$  in the same time slot of  $\sigma$ . Furthermore, the nodes of  $V_1$  require at least  $\chi(G)$  different time slots in  $\sigma$ , 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 construction schedule  $\sigma^*$  for  $G'$ , which consists of exactly  $\chi(G) + n$  time slots. Each of the first  $n$  time slots of  $\sigma^*$  contains exactly one node of  $V_2$ ; note that each of these nodes (apart from the first one) can be born by an earlier node of  $V_2$ . In each of the following  $\chi(G)$  time 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  $p$  nodes; these  $p$  nodes can be born by exactly  $p$  of the universal nodes of  $V_2$  (which have previously appeared in  $\sigma^*$ ). This completes the construction schedule  $\sigma^*$ . Since  $\sigma^*$  has  $\chi(G) + n$  time slots, it follows that  $\kappa(G') \leq \chi(G) + n$ .  $\square$

**Theorem 7.** *Let  $\varepsilon > 0$ . If there exists a polynomial-time algorithm, which, for every graph  $G$ , computes a  $n^{1-\varepsilon}$ -approximate construction schedule (i.e., a construction schedule with at most  $n^{1-\varepsilon}\kappa(G)$  slots), then  $P=NP$ .*

*Proof.* The reduction is from the minimum coloring problem. Given an arbitrary graph  $G = (V, E)$ , we construct graph  $G' = (V', E')$  as follows: We create  $2n^2$  isomorphic copies of  $G$ , which are denoted by  $G_1^A, G_2^A, \dots, G_{n^2}^A$  and  $G_1^B, G_2^B, \dots, G_{n^2}^B$ , and we also add  $n^2$  clique graphs, each of size  $2n$ , denoted by  $C_1, C_2, \dots, C_{n^2}$ . We define  $V' = V(G_1^A) \cup \dots \cup V(G_{n^2}^A) \cup V(G_1^B) \cup \dots \cup V(G_{n^2}^B) \cup V(C_1) \cup \dots \cup V(C_{n^2})$ . Initially we add to the set  $E'$  the edges of all graphs  $G_1^A, \dots, G_{n^2}^A, G_1^B, \dots, G_{n^2}^B$ , and  $C_1, \dots, C_{n^2}$ . For every  $i = 1, 2, \dots, n^2 - 1$  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  $i = 1, \dots, n^2$ , we add to  $E'$  all edges between  $V(C_i)$  and  $V(G_i^A) \cup V(G_i^B)$ . Furthermore, for every  $i = 2, \dots, n^2$ , we add to  $E'$  all edges between  $V(C_i)$  and  $V(G_{i-1}^A) \cup V(G_{i-1}^B)$ . For every  $i = 1, \dots, n^2 - 1$ , we add to  $E'$  all edges between  $V(C_i)$  and  $V(C_{i+1})$ . For every  $i = 1, 2, \dots, n^2$  and for every  $u \in V(G_i^A)$ , we add to  $E'$  the edge  $uu'$ , where  $u' \in V(G_i^B)$  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 node  $a_i$  from each  $C_i$ . We add edges among the nodes  $a_1, \dots, a_{n^2}$  such that the resulting induced graph  $G'[a_1, \dots, a_{n^2}]$  is a graph on  $n^2$  nodes which can be generated by a line construction schedule within  $\lceil \log n^2 \rceil$  slots and with zero excess edges (see Lemma 6<sup>1</sup>). This completes the construction of  $G'$ .

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

Similarly, for every  $i = 1, \dots, n^2 - 1$ , once the nodes of  $V(G_{i+1}^A) \cup \dots \cup V(G_{n^2}^A) \cup V(G_{i+1}^B) \cup \dots \cup V(G_{n^2}^B)$  have been added to the last  $(n^2 - i)\chi(G)$  slots of  $\sigma'$ , the nodes of  $V(G_i^A) \cup V(G_i^B)$  are generated in  $\sigma'$  in  $\chi(G)$  more slots. This is possible because every node  $u \in V(G_i^A) \cup V(G_i^B)$  is a

<sup>1</sup>From Lemma 6 it follows that the line on  $n^2$  nodes can be constructed in  $\lceil \log n^2 \rceil$  slots using  $O(n^2)$  excess edges. If we put all these  $O(n^2)$  excess edges back to the line with  $n^2$  nodes, we obtain a new graph on  $n^2$  nodes with  $O(n^2)$  edges. This graph is the induced subgraph  $G'[a_1, \dots, a_{n^2}]$  of  $G'$  on the nodes  $a_1, \dots, a_{n^2}$ .

candidate node after the nodes of  $V(G_{i+1}^A) \cup \dots \cup V(G_{n^2}^A) \cup V(G_{i+1}^B) \cup \dots \cup V(G_{n^2}^B)$  have been added to slots. Indeed, for every  $w \in V(C_i)$ , we have that  $N[w] \subseteq V(G_i^A) \cup V(G_i^B) \cup V(G_{i-1}^A) \cup V(G_{i-1}^B) \cup V(C_i) \subseteq N[w]$ . That is, in total, all nodes of  $V(G_1^A) \cup \dots \cup V(G_{n^2}^A) \cup V(G_1^B) \cup \dots \cup V(G_{n^2}^B)$  are generated in the last  $n^2\chi(G)$  slots.

The remaining nodes of  $V(C_1) \cup \dots \cup V(C_{n^2})$  are generated in  $\sigma'$  in  $4n - 2 + \lceil \log n^2 \rceil$  additional slots, as follows. Initially, for every odd index  $i$  and for  $2n - 1$  consecutive slots, the node  $a_i$  of  $V(C_i)$  generates exactly one other node of  $V(C_i)$ . This is possible because for every node  $u \in V(C_i) \setminus a_i$ ,  $N[u] \subseteq V(C_i) \cup V(C_{i-1}) \cup V(C_{i+1}) \subseteq N[a_i]$ . Afterwards, for every even index  $i$  and for  $2n - 1$  further consecutive slots, the node  $a_i$  of  $V(C_i)$  generates exactly one other node of  $V(C_i)$ . That is, after  $4n - 2$  slots we remain with the induced subgraph of  $G'$  on the nodes  $a_1, \dots, a_{n^2}$ . The final  $\lceil \log n^2 \rceil$  slots of  $\sigma'$  are the ones obtained by Lemma 6. Summarizing,  $G'$  is constructed by the construction schedule  $\sigma'$  in  $k = n^2\chi(G) + 4n - 2 + \lceil \log n^2 \rceil$  slots, and thus

$$\kappa(G') \leq n^2\chi(G) + 4n - 2 + \lceil 2 \log n \rceil. \quad (2)$$

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

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

$$k \geq \frac{1}{2}n^2\chi^*. \quad (3)$$

The next inequality follows by Eq. (3) and Eq. (2):

$$\frac{1}{2}n^2\chi^* \leq k \leq n^{1-\varepsilon}\kappa(G) \leq n^{1-\varepsilon}(n^2\chi(G) + 6n),$$

and thus

$$\chi^* \leq 2n^{1-\varepsilon}\chi(G) + 12n^{-\varepsilon} \quad (4)$$

Note that, for sufficiently large  $n$  we have that  $2n^{1-\varepsilon}\chi(G) + 12n^{-\varepsilon} \leq n^{1-\frac{\varepsilon}{2}}\chi(G)$ . That is, given the  $n^{1-\varepsilon}$ -approximate construction schedule produced by algorithm  $A$ , we can compute in polynomial time a coloring of  $G$  with  $\chi^*$  colors such that  $\chi^* \leq n^{1-\frac{\varepsilon}{2}}\chi(G)$ . This is a contradiction since for every  $\varepsilon > 0$ , there is no polynomial-time  $n^{1-\varepsilon}$ -approximation for minimum coloring, unless  $P=NP$  [54].  $\square$

## 5 Discussion and Future Work

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

Finally, in this section, we begin a discussion for distributed versions of this model or possible extensions that are closer to real systems. In particular, we initiate a discussion on the distributed construction of graphs, where the nodes operate autonomously, have limited capabilities, and are executing as deterministic state machines on a finite set of states  $Q$ . We study a distributed version of *growing graphs*, which we term *distributed growing graphs*. The nodes are operating in synchronous rounds, which we hereafter call *rounds*. Starting from a singleton graph  $G_0$ , the goal is to construct a target graph  $G$  in a distributed fashion. The nodes are able to perform local computations and self-replicate (i.e., generate new nodes). Additionally, each node  $u$  has local visibility of the states of all nodes  $N_u$  within the edge activation distance  $d$  from  $u$ . However, since the nodes are anonymous, each set of nodes  $S_q \in N_u$  that are in the same state  $q$  are indistinguishable among each other. Finally, we restrict our attention to the case where the edge activation distance  $d$  is two

In each round  $i > 0$ , each node  $u \in V(G_{i-1})$  executes the following steps: initially, its state defines if it will self-replicate, or not. In the former case, upon birth of a new node  $w$ , the edge  $uw$  is by default activated. Moreover,  $w$  can activate some edges with any node  $v$  such that the distance between  $w$  and  $v$  in the “intermediate” graph  $(V_t, E_{t-1} \cup \{uw : u \xrightarrow{t} w\})$  is at most  $d$ . Finally,  $u$ , based on its state, may remove some of its adjacent edges. We now describe in more detail what are the feasible edge activations/deletions that can occur.

The node-states and activated edges determine a *configuration*  $C_i$  of  $G_i$  for an execution, and can be described by a vector of all the nodes’ states and edges’ states (i.e., the existence, or not, of an edge) between each pair of nodes in  $G_i$ . A distributed protocol  $\mathcal{D}$  is defined by three functions; the transition function  $\delta_{\text{birth}}$ , the edge activation function  $\delta_{\text{act}}$ , and the edge deletion function  $\delta_{\text{del}}$ . Each node in  $G_{i-1}$ , in round  $i$  executes in sequence the transition function and the edge deletion function. All nodes in  $G_i \setminus G_{i-1}$  (i.e., the newly born nodes), execute the edge activation and the edge deletion functions, in that order. This means that an additional bit  $b$ , which defines the functions that are executed, is 1 at the time of a nodes birth, while  $b = 0$  in subsequent rounds. For clarity, we exclude this bit from the description of the above functions.

The transition function  $\delta_{\text{birth}} : Q \rightarrow (Q \times (Q \cup \epsilon))$  determines the state updates that occur in each round, and the state of the child node, if any.  $\epsilon$  means that a new node is not generated at that round, otherwise,  $\delta_{\text{birth}}$  defines the state of the child node after a self-replication. The edge activation function  $\delta_{\text{act}} : Q \rightarrow 2^Q$  determines the edge activations of a newly born node  $w$ , and finally the edge deletion function  $\delta_{\text{del}} : Q \rightarrow 2^Q$  determines the edge deletions. In other words, the state of a newly born node determines the nodes with which they activate edges with. In particular, when a node  $w$  activates an edge with a node  $v \in N_w$  in state  $q$ , it must activate all edges with nodes that are in state  $q$  in  $N_w$ ; similarly for edge deletions. Observe that we can assume without loss of generality that, for any state  $q \in Q$ ,  $\delta_{\text{act}}(q) \cap \delta_{\text{del}}(q) = \emptyset$ , as otherwise some newly added

edges would be simultaneously removed.

As an example, let  $\delta_{\text{birth}}$  consist of the following transition rule:  $a \rightarrow (b, c)$ ,  $\delta_{\text{act}}$  consist of the following edge activation rule:  $c \rightarrow \{b, d\}$ , and  $\delta_{\text{del}}$  consists of the following edge deletion rule:  $b \rightarrow \{b\}$ . The first rule means that, whenever a node  $u$  (which is currently in state  $a$ ) generates a new node  $w$ , the state of  $u$  becomes  $b$  and the state of the new node  $w$  becomes  $c$ . The second rule means that a node  $w$  (which is currently in state  $c$ ) activates all edges with all nodes in  $N_w$  that are in one of the states  $\{b, c\}$ . Finally, the third rule means that a node  $u$  (which is currently in state  $b$ ) removes all its existing edges with nodes which are currently in the same state  $b$ . A step of an execution of this protocol is shown in Figure 7.



Figure 7: The configuration  $C_i$  of  $G_i$  is shown in the first figure and  $C_{i+1}$  of  $G_{i+1}$  in the second figure. Labels represent the states of the corresponding nodes. The nodes that were in state  $a$  in  $C_i$  generated new nodes in state  $c$ , and updated their state to  $b$ .  $\delta_{\text{act}} : c \rightarrow \{b, d\}$  activated edges between the new nodes in state  $c$  and the nodes that are in state  $b$  or  $d$  in  $C_{i+1}$ , while  $\delta_{\text{del}} : b \rightarrow \{b\}$  deleted the edge between the nodes in state  $b$  in  $C_{i+1}$ .

Similarly to the centralised case of the problem, the goal is again to design protocols that *efficiently* construct a given target graph, where the two main objectives are to minimize the number of time slots to construct the target graph and the number of excess edges. In the *distributed growing graph* model, additional complexity measures may be defined. Two complexity measures that are usually considered in distributed computation models are the *local memory* of the entities and the *communication complexity*. Observe that the memory limitation that we consider renders impossible the detection of termination, therefore we require  $G_i$  to belong to the target graph class after some finite time. However, protocols with a non-constant set of states can enable termination.

In all of our distributed protocols, for ease of presentation, we split the construction into two phases, where the first phase  $\Pi_1$  constructs a graph which we call *seed*, while the second phase  $\Pi_2$  constructs the target graph, having as starting graph the seed that was constructed in the previous phase. Finally, the two sub-processes are combined into a single protocol  $(\Pi_1, \Pi_2)$ .

**Distributed cycle construction:** The seed graph of our *distributed cycle* protocol is a triangle (i.e., cycle of size 3). Starting from a singleton graph, the construction of a triangle requires two rounds; in the first round, the initiator  $u$  generates a node  $w$ , and in the second round  $w$  generates a node  $v$  and connects it with  $u$  (see Protocol 1).

---

### Protocol 1 Triangle

---

▷  $Q = \{s_1, s_2, s'_1, s'_2, s'_3\}$

▷ Initial state of initiator:  $s_1$

▷  $\delta_{\text{birth}} :$

1:  $s_1 \rightarrow (s'_1, s_2)$

2:  $s_2 \rightarrow (s'_2, s'_3)$

▷  $\delta_{\text{act}} :$

1:  $s'_3 \rightarrow \{s'_1\}$

---

The second phase of our distributed cycle construction protocol, proceeds by doubling the size of the graph in every step. Starting from a triangle seed graph, a new node is added between any two consecutive nodes of the cycle in each round. This is achieved by generating nodes and connecting them with their clockwise neighbor on the cycle.

---

**Protocol 2** Cycle

---

▷  $Q = \{s_1, s_2, s_3\}$

▷ Initial configuration:  $V = \{u_{s_1}, u_{s_2}, u_{s_3}\}$ ,  $E = \{(u_{s_1}, u_{s_2}), (u_{s_2}, u_{s_3}), (u_{s_3}, u_{s_1})\}$

▷  $\delta_{\text{birth}}$  :

- 1:  $s_1, s'_1 \rightarrow (s_1, s'_2)$
- 2:  $s_2, s'_2 \rightarrow (s_3, s'_1)$
- 3:  $s_3, s'_3 \rightarrow (s_2, s'_3)$

▷  $\delta_{\text{act}}$  :

- 1:  $s_1, s'_1 \rightarrow \{s_2\}$
- 2:  $s_2, s'_2 \rightarrow \{s_3\}$
- 3:  $s_3, s'_3 \rightarrow \{s_1\}$

▷  $\delta_{\text{del}}$  :

- 1:  $s_1 \rightarrow \{s_3\}$
  - 2:  $s_2 \rightarrow \{s_1\}$
  - 3:  $s_3 \rightarrow \{s_2\}$
- 

Observe that the node-states of Protocol 1 are  $s'_i$ ,  $1 \leq i \leq 3$  when the construction ends, while the input to Protocol 2 is a triangle with states  $s_i$ ,  $1 \leq i \leq 3$ . By utilizing an  $O(1)$  size counter in each node we can map the state of each node of the triangle to the correct input state of Protocol 2 when the construction of the triangle ends. For clarity, we haven't included these counters to our protocols.

**Distributed 4-regular graph construction:** Our second protocol constructs a 4-regular graph with a Hamiltonian cycle. In particular, the target graph forms a cycle  $C$ , where each node is additionally connected to the two nodes that are in distance 2 from them in  $C$ . The seed graph of our protocol is a graph with the same topology of size (i.e., number of nodes) six.

---

**Protocol 3** 4-regular graph of size six

---

▷  $Q = \{s_1, s_2, s'_1, s'_2, s'_3\}$

▷ Initial state of initiator:  $e_1$

▷  $\delta_{\text{birth}}$  :

- 1:  $e_i \rightarrow (e'_i, o_i)$ ,  $1 \leq i \leq 2$
- 2:  $o_i \rightarrow (o'_i, e_{i+1})$ ,  $1 \leq i \leq 2$
- 3:  $e_3 \rightarrow (e'_3, o'_3)$

▷  $\delta_{\text{act}}$  :

- 1:  $e_2 \rightarrow \{e'_1\}$
  - 2:  $e_3 \rightarrow \{e'_1, e'_2, o'_1\}$
  - 3:  $o_2 \rightarrow \{e'_1, o'_1\}$
  - 4:  $o_3 \rightarrow \{e'_1, o'_1, o'_2\}$
- 

The above protocol constructs the graph of Figure 8a, with two additional edges between the node-states  $(e'_1, o'_2)$  and  $(o'_1, e'_3)$ . Similarly to our distributed cycle protocol, we can utilize  $O(1)$  counters to remove these edges when the construction ends, and a map function to map the states of the output of Protocol 3 to the input of Protocol 4.

Our 4-regular graph construction protocol operates by doubling the size of the graph every two rounds. We call these pairs of rounds a *phase*, and  $G_p$  the graph at the end of phase  $p$ . During each phase, all nodes of  $G_{p-1}$  generate exactly one new node. Half of them (nodes in state  $o_i$ ) self-replicate in odd rounds and the rest of them (nodes in state  $e_i$ ) self-replicate in even rounds. The idea is that in each phase, a new node is added between any two consecutive nodes of the (Hamiltonian) cycle  $C$ , the corresponding edges with the nodes in distance 1 and 2 of the cycle are

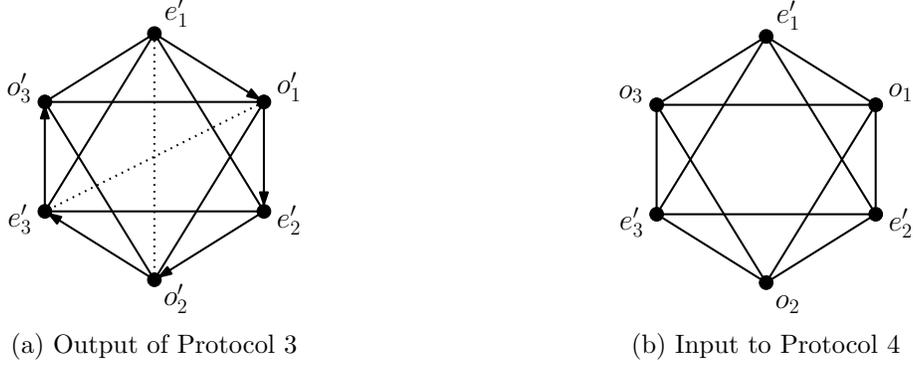


Figure 8: In (a), the labels correspond to the node-states at the end of execution of Protocol 3. The arrows represent node generations and the dotted lines are excess edges. The graph in (b) corresponds to the input of our 4-regular graph construction protocol (Protocol 4).

activated, while the rest of them (i.e., edges in distance three or more in  $C$ ) are removed.

---

#### Protocol 4 4-regular graph

---

▷  $Q = \{e_i, e'_i, e''_i, o_i, o'_i\}, 1 \leq i \leq 3$

▷ Initial configuration: the graph of Figure 8b

▷  $\delta_{\text{birth}}$  :

- 1:  $o'_i \rightarrow (o_i, \cdot), 1 \leq i \leq 3$
- 2:  $e'_i \rightarrow (e_i, \cdot), 1 \leq i \leq 3$
- 3:  $e''_i \rightarrow (e'_i, \cdot), 1 \leq i \leq 3$
- 4:  $e_1 \rightarrow (e'_1, o_1)$
- 5:  $e_2 \rightarrow (e'_2, o_3)$
- 6:  $e_3 \rightarrow (e'_3, o_2)$
- 7:  $o_1 \rightarrow (e''_1, o'_2)$
- 8:  $o_2 \rightarrow (e''_2, o'_1)$
- 9:  $o_3 \rightarrow (e''_3, o'_3)$

▷  $\delta_{\text{act}}$  :

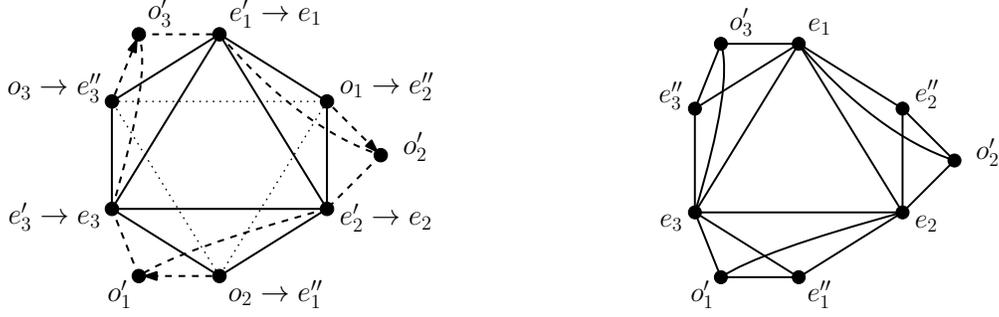
- 1:  $o_1 \rightarrow \{o_2, o_3, e'_2\}$
- 2:  $o_2 \rightarrow \{o_1, o_3, e'_3\}$
- 3:  $o_3 \rightarrow \{o_1, o_2, e'_1\}$
- 4:  $o'_1 \rightarrow \{e_2, e_3\}$
- 5:  $o'_2 \rightarrow \{e_1, e_2\}$
- 6:  $o'_3 \rightarrow \{e_1, e_3\}$

▷  $\delta_{\text{del}}$  :

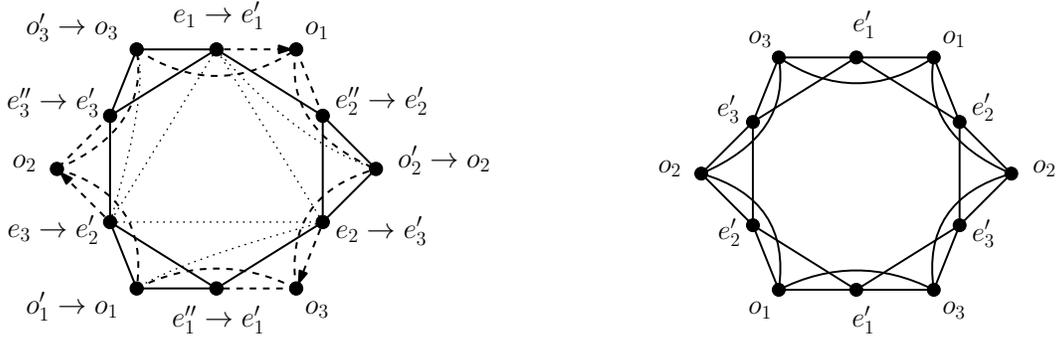
- 1:  $e_1 \rightarrow \{e_2\}$
  - 2:  $e_2 \rightarrow \{e_3\}$
  - 3:  $e_3 \rightarrow \{e_1\}$
  - 4:  $e'_1 \rightarrow \{o_2\}$
  - 5:  $e'_2 \rightarrow \{o_3\}$
  - 6:  $e'_3 \rightarrow \{o_1\}$
  - 7:  $e''_1 \rightarrow \{e''_2\}$
  - 8:  $e''_2 \rightarrow \{e''_3\}$
  - 9:  $e''_3 \rightarrow \{e''_1\}$
- 

Primed node-states indicate the number of rounds that these nodes remain idle (i.e., nodes in  $e''_i$  do not generate nodes for two rounds, and node in  $e'_i$  and  $o'_i$  do not generate nodes for one round). Let  $C_p$  be the ordered set of node-states of the Hamiltonian cycle of graph  $G_p$ , and  $C_0 = \{e'_1, o_1, e'_2, o_2, e'_3, o_3\}$ . Then  $C_p = \{e'_1, o_1, e'_2, o_2, e'_3, o_3\}^{(p+1)}, \forall p \geq 0$  (i.e.,  $|C_p| = 6(p+1)$ ). Two execution steps of this protocol are presented in Figure 9.

In this section, we considered an extension of the growing graph model, where the nodes operate autonomously, have limited capabilities, and are executing as deterministic state machines on a finite set of states  $Q$ . Other interesting extensions include models that are even closer to real life systems, for example, models where each node decides to generate another node based on its neighborhood and a random variable. This randomness encapsulates more closely the behavior of actual cells where every cell replicates based on a stochastic process and external signals from the organism.



(a) During the first round of each phase, all nodes in states  $o_i$  self-replicate. The graph on the right part is the result at the end of the first round.



(b) During the second round of each phase, all nodes in states  $e_i$  self-replicate. The graph on the right part is the result at the end of the second round of the first phase.

Figure 9: First phase of Protocol 4. Dashed arrows represent node generations, dashed lines represent edges that are activated according to  $\delta_{\text{act}}$ , and dotted lines correspond to edges that are removed according to  $\delta_{\text{del}}$ .

## References

- [1] Hugo A. Akitaya, Esther M. Arkin, Mirela Damian, Erik D. Demaine, Vida Dujmović, Robin Flatland, Matias Korman, Belen Palop, Irene Parada, André van Renssen, and Vera Sacristán. Universal reconfiguration of facet-connected modular robots by pivots: The  $O(1)$  musketeers. *Algorithmica*, 83(5):1316–1351, 2021.
- [2] 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.
- [3] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed computing*, 18(4):235–253, 2006.
- [4] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms (TALG)*, 3(4):37–es, 2007.
- [5] Hans-Jürgen Bandelt and Erich Prisner. Clique graphs and helly graphs. *Journal of Combinatorial Theory, Series B*, 51(1):34–45, 1991.

- [6] Kenneth A Berman. Vulnerability of scheduled networks and a generalization of menger’s theorem. *Networks: An International Journal*, 28(3):125–134, 1996.
- [7] A. Brandstädt, F. F. Dragan, and F. Nicolai. Lexbfs-orderings and powers of chordal graphs. *Discrete Mathematics*, 171:27—42, 1997.
- [8] Sebastian Buß, Hendrik Molter, Rolf Niedermeier, and Maciej Rymar. Algorithmic aspects of temporal betweenness. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 2084–2092, 2020.
- [9] Victor Chepoi. On distance-preserving and domination elimination orderings. *SIAM Journal on Discrete Mathematics*, 11(3):414—436, 1998.
- [10] Derek G. Corneil, Barnaby Dalton, and Michel Habib. LDFS-based certifying algorithm for the minimum path cover problem on cocomparability graphs. *SIAM Journal on Computing*, 42(3):792—807, 2013.
- [11] Derek G. Corneil and Richard Krueger. A unified view of graph searching. *SIAM Journal on Discrete Mathematics*, 22(4):1259–1276, 2008.
- [12] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. LBFS orderings and cocomparability graphs. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 883–884, 1999.
- [13] Alejandro Cornejo, Anna Dornhaus, Nancy Lynch, and Radhika Nagpal. Task allocation in ant colonies. In *International Symposium on Distributed Computing (DISC)*, pages 46–60. Springer, 2014.
- [14] Zahra Derakhshandeh, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. Universal coating for programmable matter. *Theoretical Computer Science*, 671:56–68, 2017.
- [15] Giuseppe A Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Computing*, 33(1):69–101, 2020.
- [16] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.
- [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] Ofer Feinerman and Amos Korman. The ants problem. *Distributed Computing*, 30(3):149–168, 2017.
- [20] Sándor P Fekete, Robert Gmyr, Sabrina Hugo, Phillip Keldenich, Christian Scheffer, and Arne Schmidt. Cadbots: Algorithmic aspects of manipulating programmable matter with finite automata. *Algorithmica*, 83(1):387–412, 2021.

- [21] Robert Gmyr, Kristian Hinnenthal, Christian Scheideler, and Christian Sohler. Distributed monitoring of network properties: The power of hybrid networks. In *44th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 137:1–137:15, 2017.
- [22] Seth Copen Goldstein, Jason D Campbell, and Todd C Mowry. Programmable matter. *Computer*, 38(6):99–101, 2005.
- [23] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*. Elsevier, 2nd edition, 2004.
- [24] Thorsten Götte, Kristian Hinnenthal, and Christian Scheideler. Faster construction of overlay networks. In *International Colloquium on Structural Information and Communication Complexity*, pages 262–276. Springer, 2019.
- [25] Elliot Hawkes, B An, Nadia M Benbernou, H Tanaka, Sangbae Kim, Erik D Demaine, D Rus, and Robert J Wood. Programmable matter by folding. *Proceedings of the National Academy of Sciences*, 107(28):12441–12445, 2010.
- [26] 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.
- [27] Shuji Kijima, Nobutaka Shimizu, and Takeharu Shiraga. How many vertices does a random walk miss in a network with moderately increasing the number of vertices? In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 106–122, 2021.
- [28] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 513–522, 2010.
- [29] Min Chih Lin, Francisco J. Soullignac, and Jayme Luiz Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *Theoretical Computing Science*, 426:75–90, 2012.
- [30] Nancy Lynch, Cameron Musco, and Merav Parter. Neuro-RAM unit with applications to similarity testing and compression in spiking neural networks. In *31st International Symposium on Distributed Computing (DISC)*, volume 91, pages 33:1–33:16, 2017.
- [31] Michael Andrew McEvoy and Nikolaus Correll. Materials that couple sensing, actuation, computation, and communication. *Science*, 347(6228), 2015.
- [32] George B. Mertzios and Derek G. Corneil. A simple polynomial algorithm for the longest path problem on cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 26(3):940–963, 2012.
- [33] George B Mertzios, Othon Michail, and Paul G Spirakis. Temporal network optimization subject to connectivity constraints. *Algorithmica*, 81(4):1416–1449, 2019.
- [34] George B. Mertzios, Andre Nichterlein, and Rolf Niedermeier. A linear-time algorithm for maximum-cardinality matching on cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 32(4):2820–2835, 2018.
- [35] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

- [36] 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.
- [37] Othon Michail, George Skretas, and Paul G Spirakis. Distributed computation and reconfiguration in actively dynamic networks. In *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC)*, pages 448–457, 2020.
- [38] Othon Michail and Paul G Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29(3):207–237, 2016.
- [39] János Neumann, Arthur W Burks, et al. *Theory of self-reproducing automata*, volume 1102024. University of Illinois press Urbana, 1966.
- [40] Regina O’Dell and Rogert Wattenhofer. Information dissemination in highly dynamic graphs. In *Proceedings of the 2005 joint Workshop on Foundations of Mobile Computing*, pages 104–110, 2005.
- [41] Christos H Papadimitriou. Random projection in the brain and computation with assemblies of neurons. In *10th Innovations in Theoretical Computer Science Conference*, 2019.
- [42] Benoit Piranda and Julien Bourgeois. Geometrical study of a quasi-spherical module for building programmable matter. In *Distributed Autonomous Robotic Systems*, pages 387–400. Springer, 2018.
- [43] Tim Poston. *Fuzzy Geometry*. PhD thesis, University of Warwick, 1971.
- [44] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266—283, 1976.
- [45] Paul WK Rothmund. Folding dna to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.
- [46] Paul WK Rothmund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the 32nd annual ACM Symposium on Theory of Computing (STOC)*, pages 459–468, 2000.
- [47] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.
- [48] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13:566—579, 1984.
- [49] Alan Mathison Turing. The chemical basis of morphogenesis. *Bulletin of mathematical biology*, 52(1):153–197, 1990.
- [50] Leslie G Valiant. *Circuits of the Mind*. Oxford University Press on Demand, 2000.
- [51] 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.

- [52] Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable dna self-assembly. *Nature*, 567(7748):366–372, 2019.
- [53] 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.
- [54] David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3:103–128, 2007.