We study graph realization problems from a distributed perspective. The problem is naturally applicable to the distributed construction of overlay networks that must satisfy certain degree or connectivity properties, and we study it in the node capacitated clique (NCC) model of distributed computing, recently introduced for representing peer-to-peer networks.

We focus on two central variants, degree-sequence realization and minimum threshold-connectivity realization. In the degree sequence problem, each node \( v \) is associated with a degree \( d(v) \), and the resulting degree sequence is realizable if it is possible to construct an overlay network in which the degree of each node \( v \) is \( d(v) \). The minimum threshold-connectivity problem requires us to construct an overlay network that satisfies connectivity constraints specified between every pair of nodes.

Overlay network realizations can be either explicit or implicit. Explicit realizations require both endpoints of any edge in the realized graph to be aware of the edge. In implicit realizations, on the other hand, at least one endpoint of each edge of the realized graph needs to be aware of the edge.

The main realization algorithms we present are the following.

- An \( \tilde{O}(\min\{\sqrt{m}, \Delta\}) \) time algorithm for implicit realization of a degree sequence. Here, \( \Delta = \max_v d(v) \) is the maximum degree and \( m = (1/2) \sum_v d(v) \) is the number of edges in the final realization.
- \( \tilde{O}(\Delta) \) time algorithm for an explicit realization of a degree sequence. We first compute an implicit realization and then transform it into an explicit one in \( \tilde{O}(\Delta) \) additional rounds.
- An \( \tilde{O}(\Delta) \) time algorithm for the threshold connectivity problem that obtains an explicit solution and an improved \( \tilde{O}(1) \) algorithm for implicit realization when all nodes know each other’s IDs. These algorithms are 2-approximations w.r.t. the number of edges.

We complement our upper bounds with lower bounds to show that the above algorithms are tight up to factors of \( \log n \). Additionally, we provide algorithms for realizing trees and an \( \tilde{O}(1) \) round algorithm for approximate degree sequence realization.
1 Introduction

Graph Realization problems, which deal with constructing graphs that satisfy certain specified properties, have been studied quite extensively for over half a century, focusing on problems related to realizing graphs with specified degrees [10, 18, 20], as well as other properties, like connectivity and flow [13, 14, 15, 16] or eccentricities [9, 23].

The most prominent realization problems deal with degree sequences. A (typically non-increasing) sequence of non-negative numbers \( D = (d_1, d_2, \ldots, d_n) \) is said to be realizable or graphic if there is a graph on \( n \) nodes whose sequence of degrees matches \( D \). By the handshaking lemma, dating back to Euler’s work on the Konigsburg bridges problem [11], we know that if \( D \) is graphic, then \( \sum_i d_i \) must be even. In 1960, Erdős and Gallai [10] gave a complete characterization, showing that \( D \) is graphic iff \( \sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k) \) for every \( k \in [1, n] \).

Havel [20] and Hakimi [18] independently gave a recursive algorithm that can determine whether a given \( D \) is graphic and compute a realizing graph when it is graphic. Being constructive, their method has played a crucial role in many subsequent studies, and ours is no exception.

Over the last two decades, peer-to-peer (P2P) networks have developed as a versatile and effective platform for cooperative distributed computations. Research on P2P has lead to ideas that have become crucial in a variety of contexts, ranging from fully decentralized applications like blockchain networks to more controlled contexts like Akamai’s network services [32]. Overlay construction is an important P2P component, involving the formation of new links – so called overlay links – that comprise an overlay network tailored to benefit P2P applications. In the typical scenario, starting from some basic network state, the nodes in a P2P network must interact with each other in a fully decentralized manner and form an overlay network to be used for specific purposes.

The constructed overlay network \( G \) is often required to possess certain desirable properties. A common requirement is that \( G \) be of bounded degree, so that the overhead for network formation and maintenance at each node is bounded. Additionally, one can envision a variety of other desired properties that the overlay \( G \) should possess, like bounded diameter, well-connectedness, flow guarantees, tolerance to both benign and malicious failures, and so on.

Note, however, that such overlay constructions can be viewed as (distributed) graph realization problems. This natural connection makes it plausible that ideas from graph realization theory may lend interesting new techniques allowing us to build better overlay networks. Conversely, the endeavor to build useful overlay networks is expected to pose new theoretical challenges that are likely to enrich graph realization theory. We hope that our work will initiate this new synergy between these two areas that, to the best of our knowledge, has not been formally explored in the past.

Towards the goal of formulating and studying distributed graph realization problems, we employ the node capacitated clique (NCC) model [3] that captures several key aspects of P2P networks. In this model, we have \( n \) nodes \( V \) with unique identifiers called IDs. Any node \( u \) can send messages of bounded size to any other node \( v \) provided \( u \) knows \( v \)'s ID; we can think of \( v \)'s ID as its IP address. In this sense, the NCC model is somewhat similar to the congested clique (CC) model [22, 24]. However, in the interest of being useful in the P2P context, NCC limits each node to send and receive a bounded number of messages, which, interestingly, makes NCC quite distinct from CC.

In the first paper that introduced NCC [3], node IDs were assumed to be common knowledge. This knowledge of all other IDs corresponds roughly to the \( KT_1 \) version of the CONGEST model (cf. [28]) and for this reason we call it the NCC\(_1\) model. One may argue that NCC\(_1\) is somewhat impractical in the P2P context because peers are highly unlikely to have so much knowledge about other nodes. To address this, we introduce a variant of NCC that corresponds to the \( KT_0 \) version of the CONGEST model, which we call NCC\(_0\), that limits the number of IDs known to each node. The directed graph on \( V \) in which the (out) neighbors of each \( v \) are the nodes whose IDs were known to \( v \) at the start is called the initial knowledge graph \( G_{i,v} \). In our case, we assume \( G_{i,v} \) is a path of \( n \) nodes. Since NCC\(_0\) is more restrictive, algorithms designed for it will also work in NCC\(_1\). Consequently, we focus on algorithms for NCC\(_0\) (unless stated otherwise). Further details of the model are described in Section 2.

Problem Statements. We say that an overlay graph \( G = (V, E) \) is constructed if, for every \( e = (u, v) \in E \), at least one of the endpoints is aware of the ID of the other and also aware that \( e \in E \). We say that the overlay graph is explicit if, for every edge in the graph, both endpoints are aware of the edge. Otherwise, the overlay is said to be implicit. In this paper, we focus on distributed realization problems in which, from a given initial knowledge graph and some other required input parameters, we are to construct an overlay graph that satisfies certain requirements. We study both explicit and implicit versions of the following two realization problems.

Degree Realization: Each node \( v \) in the distributed network knows its required degree \( d(v) \). The goal is to construct a realizing graph (if it exists). Formally, our input is a vector \( D = (d_1, d_2, \ldots, d_n) \) such that each \( d_i \) is only known to the corresponding node \( v_i \). The required output is an overlay graph in which every \( v_i \) has degree \( d_i \) if \( D \) is realizable; otherwise, at least one node outputs Unrealizable.
**Connectivity Threshold Realization:** The local edge connectivity threshold of two nodes \(u\) and \(v\) captures the minimum number of edge disjoint paths required between the nodes \(u\) and \(v\). In the connectivity realization problem, each node \(v\) in the distributed network is provided with a vector that captures the required minimum local edge connectivity threshold (denoted by \(\sigma(u, v)\)) to every other node \(u\). The goal is then to compute an overlay graph \(G\) with as few edges as possible so that any two nodes \(u, v\) in \(G\) satisfy the edge-connectivity relation \(\text{Conn}_G(u, v) \geq \sigma(u, v)\).

For this problem, we primarily focus on an approximate solution. In particular, we ensure that the number of edges in the overlay network is larger by at most twice that of the optimal realization.

### 1.1 Our Contributions

We present a number of new algorithms. Unless stated otherwise, these are randomized (Las Vegas) algorithms, whose running time bounds hold with high probability (w.h.p.)\(^2\). For the following results, \(m\) is the number of edges, and \(\Delta\) denotes the maximum degree in the given degree sequence.

1. For the distributed degree sequence realization problem, we provide an \(\tilde{O}(\min(\sqrt{m}, \Delta))\) time\(^3\) algorithm that produces an implicit realization of the given graphic degree sequence. We then adapt this algorithm to provide an \(\tilde{O}(\Delta)\) time algorithm for explicit realizations. We also study tree realizations, and present an algorithm for implicit realization in \(\tilde{O}(1)\) rounds. Furthermore, we optimize the diameter of the realized tree.
2. We give an \(\tilde{O}(1)\) round algorithm for implicit connectivity threshold realizations in the NCC\(_1\) model that uses at most twice the optimal number of edges needed for satisfying the connectivity threshold requirements. For NCC\(_0\) model, we give an \(\tilde{O}(\Delta)\) round explicit realization algorithm.
3. All our algorithms are tight up to factors of \(\log n\) in the NCC\(_0\) model. Specifically, we show that for implicit realizations of degree sequences there are instances that require at least \(\tilde{\Omega}(\sqrt{m})\) rounds and other instances that require \(\tilde{\Omega}(\Delta)\) rounds in NCC\(_0\). In comparison, we also show that every instance of explicit realization requires at least \(\tilde{\Omega}(\Delta)\) rounds in NCC\(_0\).
4. To facilitate the design of algorithms in NCC\(_0\), we provide some algorithmic primitives that may be of independent interest. First, we show that the nodes can be arranged in the form of a balanced binary tree in deterministic \(O(\log n)\) time. Furthermore, they can be rearranged to form a path sorted according to some parameter known locally to each node in deterministic \(O(\text{polylog}(n))\) rounds. Finally, we show how a series of primitives presented in [3] for NCC\(_1\) can be adapted to work in NCC\(_0\).

### 1.2 Related Work

A variety of graph realization problems have been studied in the literature. For the problem of realizing degree sequences, Havel and Hakimi [18, 20] independently came up with the recursive algorithm that forms the basis for our distributed algorithm. Non-centralized versions of realizing degree sequences have also been studied, albeit to a lesser extent. Arikati and Maheshwari [1] provide an efficient technique to realize degree sequences in the PRAM model. To the best of our knowledge, graph realization problems have not been explored in the distributed setting.

Other graph realization problems studied are eccentricities [9, 23], connectivity [14, 15], degree interval sequences (cf.[7] and references therein), and more (cf. [6, 8]).

Overlays in distributed settings have been well studied, as they provide structure and stability which are best exemplified by structures such as Chord [31], CAN [29], and Skip Graphs [2]. They can also be used to handle dynamism and faults. For example, Fiat and Saia [12] introduced an overlay using butterfly networks that is tolerant of faults. This was used in [3, 4] to provide a low diameter structure that is also easily addressable. Detailed surveys of various overlays and their properties can be found in [25, 26].

Our work uses the NCC model [3] for P2P networks. This model is similar to the CC model [24], introduced well over a decade ago and studied extensively since then (see [22] and references therein).

**Organization.** In section 2 we formally define the NCC model of communication. Then, to lay the groundwork for our algorithms, Section 3 presents a series of primitives along with a brief description of the Havel-Hakimi algorithm. Our main contributions are in sections 4 through 7. Section 4 presents algorithms for realizing general graphs given graphic...
degree sequences. The special case of trees is treated in Section 5. Section 6 studies the problem of connectivity threshold realization. Our lower bounds are presented in Section 7. Finally, in Section 8 we conclude.

2 The NCC model of communication

We consider the node capacitated clique model (NCC) recently introduced in SPAA’19 [3]. The NCC comprises $n$ nodes $V = \{v_1, v_2, \ldots, v_n\}$ that can communicate with each other via synchronous message passing. Each node is uniquely identified by an ID drawn from $[1,n^c]$ for some fixed $c \gg 1$. A node can send at most $O(\log n)$ messages of size $O(\log n)$ bits each per round. However, in order for a node $u$ to send a message to another node $v$, $u$ must know the ID of $v$. (Intuitively, $v$’s ID can be viewed as its IP address.)

Since inter-node communication is significantly slower than local computation at the nodes, our focus is on minimizing the round complexity (a.k.a. time complexity), which measures the number of synchronous communication rounds it takes an algorithm to terminate. Hence, the NCC model allows nodes to perform an unbounded amount of local computation per round. However, we emphasize that in our algorithms, the computation is bounded by some polynomial in $n$.

We distinguish the NCC setting into two variants, NCC$_0$ and NCC$_1$, based on the IDs initially known to the nodes. In NCC$_1$, which matches the NCC model introduced in [3], all nodes have full knowledge of each others’ IDs. Thus, w.l.o.g., we can assume that the IDs are in $[1,n]$. Thus, this version of the NCC is similar to the $K_1T_1$ variant of the CC model [24], except with a bound on the number of messages that can be sent/received at a node in every round. In NCC$_0$, on the other hand, each node only knows the IDs of a few other nodes. Formally, each node knows only the IDs of its neighbors in some directed initial knowledge graph $G_k = (V,E_k)$, such that a pair $(u,v) \in E_k$ iff $u$ knows $v$’s ID at the beginning. For concreteness, in this paper we assume that $G_k$ is a directed path consisting of the $n$ nodes arranged in some arbitrary order$^4$.

Remark. Any algorithm that can be executed in the NCC$_0$ model can be executed in the NCC$_1$ model without any increase in its time complexity. Thus, unless stated otherwise, all our algorithms are designed for NCC$_0$.

3 Preliminaries

Sections 3.1 and 3.2 describe several fundamental structural and computational primitives in NCC$_0$ that are used quite extensively later on. Then, Section 3.3 briefly describes the classical (sequential) Havel-Hakimi algorithm [18, 20], which serves as a basis for many of our algorithms.

3.1 Structural Primitives

Structural primitives deal with arranging the nodes in suitable ways. Specifically, we show how to connect the nodes in the form of a tree, or linearly, sorted by some parameter of the nodes. Recall that the initial knowledge graph $G_k$ in our model is a directed path. In one round, the directed path can be converted into an undirected (but ordered) path, by having $u$ send its ID to $v$, for each edge $(u,v)$ in $G_k$. We say that the path is ordered because, for each $(u,v)$ in the initial knowledge graph, $u$ can remember $v$ as its successor and $v$ can remember $u$ as its predecessor.

3.1.1 Balanced Binary Search Tree

Our goal is to rearrange the nodes in the form of a balanced binary search tree of height at most $O(\log n)$. The tree that is formed must be a search tree in the sense that, for every node $u$ in the tree, the nodes in the left subtree must appear earlier (i.e., precede $u$) in $G_k$ and nodes in the right subtree must appear later (i.e., succeed $u$) in $G_k$. As a warm up, we first present a simple and straightforward approach to building a balanced binary tree that will not be a search tree. To build the balanced binary tree, we exploit the fact that the nodes are arranged in a path, and that in one round, each node can learn the ID of its neighbors’ neighbors (when present). This can be accomplished by having every node send each of its neighbors’ addresses to the other neighbor. With this ability to learn one’s neighbors’ neighbor, we can decompose a path into two paths comprising nodes in odd and even positions, respectively. Now our algorithm can be described succinctly in the following recursive manner. Initially, there is a single path, but as the algorithm progresses, several paths will be created and the recursive step, described as follows, must be applied to every path. The left-most node $r$ makes its immediate neighbor $a$ its left child; $r$ then makes $a$’s other neighbor $b$ its right child. Then, $r$ removes itself from the path and the path is decomposed into two paths comprising nodes in odd and even positions, with $a$ and $b$ being the left-most and rightmost nodes in the two paths. This process is then repeated recursively (and in parallel) in the two paths that are created, and the recursion terminates when empty paths are created. In $O(\log n)$ rounds, all recursive calls would have terminated because the length of the paths halve in each recursive call.

$^4$We choose to make $G_k$ a path for ease and clarity in exposition. Typically, $G_k$ will be a low-degree graph.
The binary tree on \( G_k \)

Figure 1: The balanced binary tree built on \( G_k \) using the recursive construction. For any path, \( r, a, \) and \( b \) indicates the parent and its left and right child respectively.

We now provide a more elaborate algorithm to produce a balanced binary search tree. We first design a structure \( L \) and then perform a controlled breadth first search that results in a balanced binary tree. Our construction is fully distributed and deterministic. The structure \( L \) comprises \( \lceil \log n \rceil + 1 \) levels \((L_0, L_1, \ldots, L_{\lceil \log n \rceil})\). Each level \( i, 0 \leq i \leq \lceil \log n \rceil \), comprises up to \( 2^i \) paths (set of nodes with structure similar to \( G_k \), but undirected). \( L_0 \) has just one path, which is the undirected form of \( G_k \). Subsequently, each path \( \ell \) at level \( L_{i-1} \) produces up to two paths \( \ell_0 \) and \( \ell_1 \) at level \( L_i \); \( \ell_0 \) comprises nodes at even position in \( \ell \) and \( \ell_1 \) contains the nodes in odd position. We say that path \( \ell \) is the parent path of paths \( \ell_0 \) and \( \ell_1 \), and conversely, \( \ell_0 \) and \( \ell_1 \) are the children of \( \ell \). In order to bypass the problem of nodes having to determine the parity of their positions, we do not require nodes in \( \ell_0 \) and \( \ell_1 \) to be aware of whether they are in \( \ell_0 \) or \( \ell_1 \).

To construct \( \ell_0 \) and \( \ell_1 \), each node in \( \ell \) sends its predecessor’s ID to its successor and vice versa and then each node \( v \) in \( \ell \) connects to its grand-predecessor \( p^* \) (i.e., predecessor’s predecessor) in \( \ell \) and (the similarly defined) grand-successor \( s^* \) (both, only if present). Node \( p^* \) and \( s^* \) now serve as the predecessor and successor, respectively, of \( v \) in the newly formed path in level \( L_i \). After all paths in level \( L_{\lceil \log n \rceil} \) are constructed, the first node in \( G_k \) (i.e., the node with no predecessor at level \( L_0 \)) becomes the root node \( r \) and initiates a controlled BFS as described in Algorithms 1 to form a BFS tree \( T_{BFS} \).

**Theorem 1.** The fully distributed deterministic construction outlined above takes a linear arrangement of nodes as the knowledge graph \( G_k \) and produces a binary tree \( T_{BFS} \) of height at most \( \lceil \log n \rceil + 1 \) in \( O(\log n) \) rounds. Moreover, an inorder traversal of this tree yields the original \( G_k \).

**Proof.** The paths on each level are close to half the length of their parent path on the previous level, so there are clearly at most \( O(\log n) \) levels and the construction of each path takes at most \( O(1) \) parallel rounds. Thus \( L \) is constructed in \( O(\log n) \) rounds. Since, in the binary tree construction, invitations and subsequent acceptances happen in parallel, each iteration only takes \( O(1) \) rounds of height at most \( O(\log n) \), thereby making the overall running time \( O(\log n) \).

To argue the correctness of the binary tree construction, we observe the following properties of \( T_{BFS} \).

**Single Parent.** Each node has exactly one parent because it only accepts one invitation. (Once it joins \( T_{BFS} \), it stops accepting invitations.)

**Two Children.** Each node \( v \) has at most two children because it gets at most one predecessor child and one successor child.
Figure 2: The structure ℒ and the corresponding balanced binary search tree built on ℒ. The letters p and s indicate the predecessor and successor of a node in a level respectively.

Algorithm 1: A controlled BFS algorithm to arrange nodes in the form of a balanced binary tree.

Input: ℒ has been constructed and the root r knows that it’s the root.
Output: A balanced binary tree rooted at r such that an in-order traversal will yield the nodes in the input knowledge graph ordering.

/* Sets $S_p, S_s$ are maintained implicitly in the sense that each node in $S_p$ or $S_s$ knows its membership. */

1. $S_p \leftarrow S_s \leftarrow \{r\}$;
2. for $i$ from $\lceil \log n \rceil$ down to 0 do
   3. foreach $v \in S_p$ (in parallel) do
      4. if $v$ has a predecessor $p$ in level $i$ then
         5. $v$ invites $p$ as its left child in the BFS;
         6. $S_p \leftarrow S_p \setminus \{v\}$;
   7. foreach $v \in S_s$ (in parallel) do
      8. if $v$ has a successor $s$ in level $i$ then
         9. $v$ invites $s$ as its right child in the BFS;
         10. $S_s \leftarrow S_s \setminus \{v\}$;
   11. foreach node $u$ not already in the BFS tree that was invited (in parallel) do
       12. $u$ accepts the invitation from, say, $v$;
       13. $u$ connects to $v$ forming a BFS tree edge between parent $v$ and child $u$;
       14. $S_p \leftarrow S_p \cup \{u\}$;
       15. $S_s \leftarrow S_s \cup \{u\}$;

Balanced. The depth of $T_{BFS}$ is at most $O(\log n)$ (which again holds because there are only $O(\log n)$ levels in ℒ and each parent is always at a higher level than its children).
To show that \( V \) graph which is a linear arrangement of all nodes in \( G \) gets an invitation at any level higher than \( V \). The case when \( v \) is not the successor of \( v' \) at level \( \text{elev}(v) \). The node \( v' \) must have a higher elevation as both \( v \) and \( v' \) cannot have the same parity in \( t_{\text{elev}(v)} \). Thus, by our assumption, \( v' \) must have been invited, entered \( T_{BFS} \), and put itself in both \( S_p \) and \( S_s \) at level \( \text{elev}(v') \). From level \( \text{elev}(v') \) down to \( \text{elev}(v) + 1 \), \( v' \) could not have had any successor, as otherwise, \( v \) would not be its successor at level \( \text{elev}(v) \), so \( v' \) must invite \( v \) at level \( \text{elev}(v) \), contradicting our assumption.

Finally, we show that each node can compute its position in the path graph \( G_k \) in \( O(\log n) \) rounds. We first construct \( T_{BFS} \) on \( G_k \). Recall that the in-order traversal of \( T_{BFS} \) is \( G_k \), so the problem reduces to each node calculating its in-order traversal number in \( T_{BFS} \). This can be done efficiently in \( O(\log n) \) rounds [4]. The key idea is to perform a bottom up phase in which nodes learn the size of their (and their children’s) subtrees, and then a top down phase in which the nodes learn their in-order traversals. We assume that \( n \) is known, but the algorithm can be modified to work as long as we have a reasonable upper bound for it. In such situations, the exact number of nodes can be inferred from \( T_{BFS} \) as the in-order traversal number of the last node in the original ordering. Thus, once \( n \) becomes common knowledge, the median node’s ID can also be inferred in \( O(\log n) \) rounds (and flooded to all nodes if needed).

**Corollary 2.** The position of a node in a path graph can be found in \( O(\log n) \) rounds. Similarly, the address of the median node can be made common knowledge to all nodes within \( O(\log n) \) rounds.

### 3.1.2 Sorting in NCC \(_0\)

Consider the NCC \(_0\) model with knowledge graph \( G_k \) being a path of \( n \) nodes. Furthermore, let each node be assigned a unique value. We now show how to build a sorted path on those \( n \) nodes. To be more precise, we define a sorted path as a path of nodes such that the first node has the smallest value and is called the head node, the last node has the largest value and is called the tail node, and all other nodes have values greater than (resp., smaller than) their predecessor (resp., successor). Every node in a sorted path must know its predecessor and successor’s IDs (when present). In addition, for convenience, we assume that a balanced binary search tree has been constructed on the set of nodes, i.e., a tree of height that is asymptotically at most logarithmic in the number of nodes such that for every node \( v \), the values in its left (resp., right) subtree are smaller (resp., greater) than \( v \)’s value. The head node’s ID will serve as the handle for the sorted path.

Our approach is to first build a balanced binary search tree \( T \) (see Theorem 1) that may not be ordered according to node values, and then build a sorted path of the \( n \) nodes in a bottom up manner starting with the leaves. To build the sorted path, first each leaf node sends its value up to its parent. Then the parent sorts them, constructs the sorted path with itself and its children, and informs the leaves of their neighbors in the sorted path. This is then repeated at higher levels, i.e., each node \( u \) passes on the ID of the head node of the sorted path containing all nodes in \( u \)’s subtree up to its parent \( v \). Node \( v \) will receive up to two such IDs, one from each of its children. Node \( v \) then uses the merging techniques described shortly to merge the two corresponding paths (also including itself) and passes on the merged path’s head ID to its parent. This bottom-up procedure will therefore require us to go through \( O(\log n) \) levels. At each level, the nodes must merge the two sorted paths they receive from their children, for which we provide an \( O(\log^2 n) \) rounds procedure. Thus, the overall round complexity is \( O(\log^3 n) \).

We now focus on a node \( v \) that has received two head node IDs corresponding to the sorted paths \( R_1 \) and \( R_2 \) of nodes from \( v \)’s left subtree and right subtree, respectively. Node \( v \) initiates a recursive merge procedure that combines the two paths into one sorted path \( R \) and then inserts itself into the sorted path (taking advantage of the fact that the sorted
path \( R \) will include a balanced binary search tree. Thus, the output will be a sorted path containing all nodes in the subtree rooted at \( v \).

The merge procedure is defined recursively. Without loss of generality, let \( R_1 \) be the larger of the two paths (breaking ties arbitrarily) that \( v \) receives from its children in \( T \). If \( R_2 \) has just one node \( y \) in it, that singleton node \( y \) and \( v \) are just inserted into \( R_1 \) to produce the required merged sorted path \( R \). Otherwise, The median node \( x \) of \( R_1 \) is computed (see Corollary 2) and each of the two sorted paths is split into two sorted paths as follows. Path \( R_1 \) is split into \( R_1^< \) (resp., \( R_1^> \)) comprising nodes in \( R_1 \) with values less than (resp., more than) the value of \( x \). Path \( R_2^< \) and \( R_2^> \) are defined similarly. These paths can be constructed in a straightforward manner by searching for the value of \( x \) in the two paths \( R_1 \) and \( R_2 \). For example, the node in \( R_1 \) with the largest (resp., smallest) value that is less than (resp., more than) the value of \( x \) will be \( R_1^< \)’s tail (resp., \( R_1^> \)’s head) and having computed the required tails and heads, we can apply Theorem 1 to complete the construction of the required sorted paths. Thus, these sorted path constructions will take \( O(\log n) \) time. We now recursively merge \( R_1^< \) and \( R_2^> \) to form the merged path \( R^< \) comprising all nodes from \( R_1 \) and \( R_2 \) whose values are less than that of \( x \). The path \( R^> \) whose values are greater than that of \( x \) is also obtained similarly. The two paths \( R^< \) and \( R^> \) are sorted, so they can now be merged easily (with \( x \) placed in between) to form the required sorted path \( R \).

The correctness of the procedure is straightforward and similar to mergesort. Note that the recursion depth of each merge procedure will be at most \( O(\log n) \) because the sum of the nodes in the two paths at any recursive step will be at most \( 3/4 \) of the number of nodes in the two paths at the previous recursive step. This follows from our choice of \( x \) as the median element in the larger of the two paths. Thus, the depth of the recursion employed by the merge procedure is \( O(\log n) \). At each recursive step, we will be required to compute the median element (see Corollary 2) of the larger sorted path, search within the sorted paths, and build a balanced binary search tree (see Theorem 1), each requiring at most \( O(\log n) \) rounds.

**Algorithm 2:** Recursive-Merge

**Input:** A node \( v \) in the network receives two paths \( R_1 \) and \( R_2 \).

**Output:** The path \( R \), where \( R \) is created by merging (in a sorted order) of \( R_1 \) and \( R_2 \).

1. Create balanced binary search trees for both \( R_1 \) and \( R_2 \) (see Corollary 2). **If either** \( R_1 \) **or** \( R_2 \) **has only one node** \( y \) **then**
   2. Insert \( y \) in the appropriate position using the balanced binary search tree;
   3. Return \( R \).
4. **Else**
   5. Find the median of the larger path using the balanced binary trees;
   6. Use it to split \( R_1 \) and \( R_2 \) into four sub-paths \( R_1^<, R_1^>, R_2^<, R_2^> \);
      
      /* Here \( R_1^< \) (respectively \( R_1^> \)) represents the parts of path \( R_1 \) whose values are less than (respectively, greater than) that of the median. \( R_2^< \) and \( R_2^> \) are defined similarly. */
   7. Call the recursive merge procedure on the pairs \( R_1^<, R_2^< \) and \( R_1^>, R_2^> \).

**Theorem 3.** There exists an algorithm for creating a sorted path graph on \( n \) nodes of a balanced tree \( T \) in \( O(\log^3 n) \) rounds in the NCC\(_0\) (and hence also in NCC\(_1\)) model.

### 3.2 Computational Primitives

The computational primitives described next are primarily for aggregating, collecting, broadcasting and multicasting information in NCC\(_0\). The following problem variations have been presented in [3] for the NCC\(_1\) model. We briefly state how we adapt them to NCC\(_0\) and restate the results for the sake of completeness.

#### 3.2.1 Global Computational Primitives

In **global broadcast**, a token initially held by a designated leader node \( \ell \) is to be sent to all other nodes. After constructing the balanced binary tree, the leader can simply send the token to the root and the root can then send it down to all nodes.

For **global aggregation**, we first need to define a few terms. An aggregate function is a function on any given set of items \( I \). In the context of distributed computing, we may encounter the items in \( I \) in various orders, so
we use a special form of aggregate functions called *distributive aggregate functions*. An aggregate function \( f \) is distributive if there is another aggregate function \( g \) such that for every partition of \( I \) into sets \( I_1, I_2, \ldots, I_k \),
\[
 f(I) = g(f(I_1), f(I_2), \ldots, f(I_k)) .
\]
We assume that \( g \) is known whenever \( f \) is known; in fact, they are often the same (e.g., maximum and minimum). In global aggregation, a designated leader node \( \ell \) is known to all nodes, and every node \( u \) in \( V \) has an input, \( \text{value}_{u,v} \), of size at most \( O(\log n) \) bits, and is aware of a distributive aggregate function \( f \) over \( I \) that produces \( f(I) \) of size at most \( O(\log n) \). The goal is for the leader to learn \( f(\{\text{value}_{u,v}\}_{v \in V}) \).

Again, constructing a balanced binary tree, we may aggregate the values to the root using the standard convergecast technique, which will then send \( f(I) \) to \( \ell \).

**Theorem 4.** *Global broadcast and global aggregation can be performed in \( O(\log n) \) rounds.*

### 3.2.2 Global Collection

Again, a designated leader \( \ell \) is known to all nodes. For some \( A \subseteq V \) of size \( |A| = k \), each \( v \in A \) has a token \( \text{token}_{v,i} \) of size \( O(\log n) \) bits and the goal is to collect \( \{\text{token}_{v,i}\}_{v \in A} \) at \( \ell \). Using the balanced binary tree and pipelining, we get:

**Theorem 5.** *Global collection can be performed in \( O(k + \log n) \) rounds.*

### 3.2.3 Local Computational Primitives

The first local task is *local aggregation* in which we seek to aggregate data over \( g \) different aggregation groups \( A_1, A_2, \ldots, A_g \subseteq V \). Each node \( v \in A_i \) has an input, \( \text{value}_{v,i} \). The \( A_i \)'s are not necessarily disjoint, so the number of input values each \( v \) has equals the number of \( A_i \)'s that contain it. Each aggregation group \( A_i \) has an associated destination node \( t_i \) (not necessarily in \( A_i \)) that must learn \( f_i(\{\text{value}_{v,i}\}_{v \in A_i}) \) for some distributive aggregate function \( f_i \) known to all \( v \in A_i \). Each aggregation group \( A_i \) has a unique identifier \( \text{gid}_i \in [n] \) known to all its members.

Define \( L = \sum_i |A_i| \), \( \ell_1 = \max_{v \in V} \{|i \in [g] \mid v \in A_i\}| \) and \( \ell_2 = \max_{v \in V} \{|i \in [g] \mid v = t_i\}| \).

The participants of \( A_i \cup \{t_i\} \) are not required to know the other participants.

To perform the aggregation task (and other subsequently defined tasks), we need to ensure that the nodes in \( V \) can emulate a butterfly network (see [27] for definition and details) within the framework of \( \text{NC}_0 \). This can be accomplished in \( O(\log n) \) time by adapting a recursive procedure given in [4]; also see [3] for specifications of the required emulation.

**Theorem 6 (Theorem 2.2 in [3]).** *Aggregation can be performed by a randomized algorithm in \( O\left( \frac{L}{n} + \frac{\ell_1 + \ell_2}{\log n} + \log n \right) \) rounds w.h.p.*

The second primitive is *local multicasting*. This task concerns sets \( A_1, A_2, \ldots, A_g \subseteq V \) and source nodes \( \{s_1, s_2, \ldots, s_g\} \). Each multicast group \( A_i \cup \{s_i\} \) has a unique group identifier \( \text{gid}_i \), known to all its participants. Each source node \( s_i \) has a token, \( \text{token}_{s,i} \), that must reach all nodes in \( A_i \). The parameters \( L \) and \( \ell_1 \) are defined as before. Also let \( \ell_3 = \max_{v \in V} \{|i \in [g] \mid v = s_i\}| \).

**Theorem 7 (Theorem 2.5 in [3]).** *Multicast can be performed in \( O\left( \frac{L}{n} + \frac{\ell_1 + \ell_3}{\log n} + \log n \right) \) rounds.*

Finally, we consider the task of collecting tokens, where we again have groups \( A_1, A_2, \ldots, A_g \subseteq V \), and each \( v \in A_i \) has an input token \( \text{token}_{v,i} \) of size at most \( O(\log n) \). The \( A_i \)'s are not necessarily disjoint, so the number of tokens at each \( v \) equals the number of \( A_i \)'s that contain \( v \) (assuming \( v \) has a different token for each group it belongs to). Each group \( A_i \) has an associated destination node \( t_i \) (not necessarily in \( A_i \)) that must collect the tokens in \( \{\text{token}_{v,i} \mid v \in A_i\} \).

As usual, each group \( i \) has a unique group ID associated with it. For simplicity, we assume that no two groups share the same destination.

**Theorem 8.** *Token collection can be performed in \( O\left( \frac{L}{n} + \frac{\ell_3}{\log n} + \log n \right) \) rounds.*

**Proof.** The goal is to map the given token collection task into a suitable aggregation task. Our approach is to create summation groups — one group for each token — that comprises just two nodes each, namely, the source of the token and its destination. Concretely, consider \( \text{token}_{v,i} \) at \( v \in A_i \) that must reach \( t_i \). Then, we need to form an aggregation group with the singleton set \( \{v\} \) and destination \( t_i \). The challenge is to get both nodes in each group to agree on a group ID because, if we use any property of the source \( v \) (either its ID or position), the destination \( t_i \) may not know it.

\(^5\)In [3], the unique group identifier happens to be the ID of \( t_i \), but as commented in [3], any arbitrary unique identifier will suffice equally well. Some of our algorithms, however, crucially require choice of group identifiers other than the ID of \( t_i \).
Instead, we form a unique group ID by concatenating the following: the group ID of group \( i \) (in the token collection instance) and \( u_i(v) \), where \( u_i(v) \) is the in-order traversal number of \( v \) in the multicast tree [3] that is used by group \( i \), which we know how to compute from Corollary 2. Now we can apply Theorem 6 to get the required result.

### 3.3 Sequential Havel-Hakimi Algorithm

The characterization of Havel [20] and Hakimi [18] for graphic sequence can be stated concisely as follows.

**Theorem 9** (Based on [20] and [18]). A non-increasing sequence \( D = (d_1, d_2, ..., d_n) \) is graphic if and only if the sequence \( (d'_2, ..., d'_n) \) is graphic, where \( d'_j = d_j - 1 \), for \( j \in [2, d_i + 1] \), and \( d'_j = d_j \), for \( j \in [d_i + 2, n] \).

This characterization directly implies an \( O(\sum d_i) \) time sequential algorithm, known as the Havel-Hakimi algorithm, for constructing a realizing graph \( G = (V, E) \) where \( V = \{v_1, v_n\} \) and \( \text{deg}(v_i) = d_i \), or deciding that no such graph exists. The algorithm works as follows. Initialize \( G = (V, E) \) to be an empty graph on \( V \). In step \( i \): (a) remove \( d_i \) from \( D \), and set \( d_j = d_j - 1 \) for all \( j \in [i + 1, d_i + i + 1] \); (b) set the neighborhood of the node \( v_i \) to be \( \{v_{i+1}, v_{i+2}, ..., v_{i+1+d_i}\} \); (c) finally, sort the updated \( D \) (as well as associate nodes). If, at any step, \( D \) contains a negative entry, then the sequence is not realizable.

### 4 Distributed Degree Realization in Graphs

Subsection 4.1 presents an \( \tilde{O}(\min\{\sqrt{m}, \Delta\}) \) time algorithm for implicit realization, where \( \Delta \) is the maximum degree and \( m \) is the number of edges in the realizing graph. Subsection 4.2 extends this to an \( \tilde{O}(\Delta) \) time algorithm for explicit realization. Then, Subsection 4.3 presents a distributed degree realization algorithm that, allowing for small variations in the assigned degrees, obtains a realization in the scenarios where the degree sequence is not graphic. This plays a crucial role in the connectivity threshold realizations presented later in Section 6.

#### 4.1 Implicit Degree Realization in \( \tilde{O}(\min\{\sqrt{m}, \Delta\}) \) Time

We first give a high-level description of our approach to implicit degree realization. The detailed pseudocode is presented in Algorithm 3. It is a parallel version of the well known Havel-Hakimi procedure [18, 20]. Recall that the Havel-Hakimi algorithm works by ensuring that the highest degree node is satisfied first (and once satisfied, its degree is set to zero). So overall, the maximum degree in graph has decreased by at least \( \delta - 1 \).

Our algorithm proceeds in phases. Each phase operates as follows. We first sort the nodes in non-increasing order of degrees in \( O(\log^3 n) \) rounds via techniques presented in Section 3. These nodes are arranged in a path graph \( L \). Each node determines its position in \( L \) (see subsection 3.1). Let \( \delta_j \) be the value of the maximum degree of any node in the \( j \)th phase of the algorithm and \( \Delta \) denote the number of nodes of degree \( \delta_j \) in that phase. Define \( q = \max\{1, \lfloor N / \delta_j + 1 \rfloor \} \).

Then on \( L \), starting with the highest degree nodes, divide the first \( q(\delta_j + 1) \) nodes into \( q \) multicast groups, \( g_1, ..., g_q \) of \( \delta_j + 1 \) nodes each. In every group \( g_i, i \in [q] \), the leftmost node \( t_i \) (in the path graph \( L \)) multicasts its ID to nodes in \( g_i \). Thus, every other node \( v \in g_i \), \( k \), knowing \( t_i \)'s ID, forms an implicit overlay edge with \( t_i \). Note that each group \( g_i \) can use \( i \) as the group's unique identifier. The nodes \( t_i, i \in [q] \) set their degrees to zero, while all other nodes in the groups decrease their degree by 1. This mean that \( q \) nodes of degree \( \delta_j \) have now disappeared from the graph (as they have been realized and are no longer under consideration). We then re-sort the list and repeat the above procedure until all nodes have degree 0.

**Lemma 10.** The while loop in Algorithm 3 is invoked \( O(\min\{\Delta, \sqrt{m}\}) \) times (i.e., there are at most \( O(\min\{\Delta, \sqrt{m}\}) \) phases).

**Proof.** At any phase \( j \) (\( j \)th iteration of the while loop), let \( \delta_j \) be the maximum degree and \( N \) be the number of nodes of degree \( \delta_j \). We distinguish two cases.

If \( N \leq \delta_j + 1 \), then the nodes \( x_2, x_3, ..., x_{\delta_j} \) store \( ID(x_1) \) in their neighbor-list, and decrease their degree by 1. Also, \( x_1 \) is removed by setting its degree to zero. So overall, the maximum degree in graph has decreased by at least 1.

Otherwise \( (N > \delta_j + 1) \), for \( q = \lfloor N / \delta_j + 1 \rfloor \) distinct values of \( i \), the nodes \( x_{i+1}, x_{i+2}, ..., x_{i+\delta_j} \), store \( ID(x_i) \) in their neighbor-list, and decrease their degree by 1. Intuitively, we remove \( q \) stars from \( G \). Though the maximum degree may remain \( \delta_j \) after this process, the number of nodes left with degree \( \delta_j \) must be at most \( \delta_j \). This implies in the next round, \( N \) would be bounded by \( \delta_j + 1 \), so at most one additional round would be needed to eliminate \( \delta_j \).
**Algorithm 3: Distributed-Degree-Realization**

**Input:** An $n$-node network in which each node $x$ is provided with a degree $d(x)$.

**Output:** A corresponding implicit realization that satisfies the degrees assigned to the nodes.

1. $j = 0$
2. while (1) do
3. \hspace{1em} $j \leftarrow j + 1$
4. \hspace{2em} Sort the nodes in the non-increasing order of degrees; refer to the nodes as $x_1, \ldots, x_n$ such that $d(x_1) \geq d(x_2) \geq \ldots \geq d(x_n)$.
5. \hspace{2em} (After sorting, each node knows its position in $L$.)
6. \hspace{2em} Broadcast $\delta_j = d(x_1)$, the current maximum degree, to all the nodes in the network.
7. \hspace{2em} if $\delta_j \geq 1$ then
8. \hspace{3em} Aggregate and broadcast to all the nodes the value $N = \max\{i \mid d(x_i) = \delta_j\}$.
9. \hspace{3em} Let $q = \max\{1, \lfloor \frac{N}{\sqrt{m}} \rfloor\}$.
10. \hspace{3em} foreach $i \in \{\alpha(\delta_j + 1) - \delta_j \mid \alpha \in [q]\}$, in parallel do
11. \hspace{4em} Set $d(x_i) = \text{NIL}$.
12. \hspace{4em} Broadcast $ID(x_i)$ to the next consecutive $\delta_j$ successors of $x_i$, i.e. $x_{i+1}, x_{i+2}, \ldots, x_{i+\delta_j}$.
13. \hspace{4em} The nodes $x_{i+1}, \ldots, x_{i+\delta_j}$ store $ID(x_i)$ in their neighbor-list, and decrease their degrees by 1. In the process, if the degree of a node becomes negative, then it broadcasts UNREALIZABLE, and the execution terminates.
14. \hspace{2em} else exit the while loop;

Basically, in each iteration of the while loop (or in some cases with an additional iteration), the maximum degree for that phase is removed from consideration. This guarantees the number of iterations to be at most $\Delta$. Observe that, the number of nodes of degree greater than $\sqrt{m}$ in any $m$-edge graph is at most $O(\sqrt{m})$. Also, in each iteration, the degree of at least one node of maximum degree becomes zero; it follows that the number of iterations required to remove all nodes with degree $> \sqrt{m}$ is at most $O(\sqrt{m})$. For the remaining nodes with degree $\leq \sqrt{m}$, the $O(\Delta)$ bound translates to a $O(\sqrt{m})$ bound. Hence, the number of iterations is bounded by $O(\min\{\Delta, \sqrt{m}\})$. 

We next describe an $\tilde{O}(1)$ time implementation of an single phase in Algorithm 3 in the NCC model. We briefly look at the sorting, aggregation of frequency/maximum, and selective broadcasting procedures in an individual phase (steps 2-10 in Algorithm 3). The sorting in step 2 is accomplished in $O(\log^3 n)$ rounds via the sorting techniques presented in Sect. 3. This allows all nodes to be arranged in the sorted path $L$ (sorted by the decreasing order of degrees). Crucially, at the end of step 2 each node knows its position in $L$. We then accomplish step 4 via a global broadcast and inform all the nodes in the network of the value $\delta_j$. Step 6 can then be accomplished by aggregation and broadcast. First, we calculate the value of $N$ by having each node $x_i$, such that $d(x_i) = \delta_j$, aggregate towards the node $x_1$ using the group ID 1. Then $x_1$ performs a global broadcast to inform the nodes in $L$ of $N$. All of the above can be accomplished in $O(\log n)$ rounds using the global aggregation and broadcast techniques discussed in Sect. 3 (Theorem 4).

Once $N$ is known to all the nodes in the network, each node can locally compute the value of $q$. In step 7, we form $q$ distinct groups for $i \in [1, q]$. Each $i$ is a group ID for any node that is going to become a neighbor for $x_i$. Note that a node can calculate which distinct group it belongs to locally (with information about its position and $N$). Thus, for any $i$, each node $x_i$ can use $i$ as the group ID and use the multicast algorithm to broadcast its ID to nodes $x_{i+1}, x_{i+2}, \ldots, x_{i+\delta_j}$. All of this can be done in parallel. Notice that by Theorem 7, steps 5-7 can be done in $O(\log n)$ rounds. Thus, we have:

**Theorem 11.** There exists a procedure for implicitly realizing any given length-$n$ graphic sequence $D = (d_1, \ldots, d_n)$ in $\tilde{O}(\min\{\sqrt{m}, \Delta\})$ rounds, in both the NCC$_0$ and NCC$_1$ models, where $\Delta$ is the maximum degree in $D$ and $m$ is the number of edges required for the realization.

**Proof.** By lemma 10, Algorithm 3 involves at most $O(\min\{\sqrt{m}, \Delta\})$ iterations of the while loop. By the above discussion, a single iteration of the while loop (steps 2-10) can be performed in $\tilde{O}(1)$ rounds. It follows that the entire construction process takes $\tilde{O}(\min\{\sqrt{m}, \Delta\})$ rounds. 

\[\square\]
4.2 An $\tilde{O}(\Delta)$ Time Algorithm for Explicit Degree-Realization

We now describe the extension of Theorem 11 for explicit realization. We first execute Algorithm 3. At the end of its execution, each edge is stored implicitly. That is, for any edge $e = (u, v)$ that was formed, at least one of its endpoints (say $u$) is aware of the edge’s existence (and $v$’s ID). Therefore, $u$ must communicate its ID to $v$ to make the realization explicit. To accomplish this, we create a group for each $v$ such that the associated set is all nodes that must communicate their IDs to $v$. Now applying Theorem 8, we can say that

Theorem 12. There exists a procedure for explicitly realizing any given length-$n$ graphic sequence $D = (d_1, \ldots, d_n)$ in $O(m/n + \Delta/\log n + \log n)$ rounds, in both the NCC$0$ and NCC$1$ models, where $m = (1/2)\sum_i d_i$ and $\Delta = \max_i d_i$.

4.3 Approximately Realizing Non-Graphic Sequences

We next consider the case of a non-realizable degree sequence $D = (d_1, d_2, \ldots, d_n)$. An upper envelope to $D$ is a degree sequence $D' = (d'_1, d'_2, \ldots, d'_n)$ satisfying $d'_i \geq d_i$ for every $i$. A natural goal is to find a realization for an upper envelope $D'$ of low total discrepancy with $D$, defined as $\varepsilon(D, D') = \sum_{i=1}^n (d'_i - d_i)$. The question of minimizing the discrepancy was studied by Hell and Kirkpatrick in the centralized setting [21]. We present a distributed solution that provides an explicit realization of an upper envelope $D'$ for a given non-realizable sequence, with a discrepancy of at most $\sum_{i=1}^n d_i$. This only requires the following alteration to step 13 of Algorithm 3.

Step 13: The nodes $x_{i+1}, \ldots, x_{i+\delta_i}$ store $ID(x_i)$ in their neighbor-list, and decrease their degrees by 1. In the process, if degree of a node becomes negative, then it resets its degree to 0.

Let $D' = (d'_1, d'_2, \ldots, d'_n)$ be the degree-sequence of the output graph. It is easy to see that the total degree increase $\sum_{i=1}^n (d'_i - d_i)$ is bounded by $\sum_{i=1}^n d_i$. This is because when a node’s degree is reset to 0, the resorting ensures that it will again be used as a neighbor at most $d_i$ times, yielding the following.

Theorem 13. There exists a procedure for explicitly realizing, for any given length-$n$ (possibly non-graphic) sequence $D = (d_1, \ldots, d_n)$, an upper envelope $D' = (d'_1, \ldots, d'_n)$ satisfying (i) $d'_i \geq d_i$ for every $i$, and (ii) $\sum_{i=1}^n d'_i \leq 2\sum_{i=1}^n d_i$. This can be done in $\tilde{O}(\Delta)$ rounds in both the NCC$0$ and NCC$1$ models, where $\Delta$ is the maximum degree in $D$.

5 Degree-Sequence Realization in Trees

In this section, we consider the degree realization problem when we restrict our realizations to trees. We refer to this as the tree-realization problem. Note that a degree sequence $(d_1, d_2, \ldots, d_n)$ is realizable as a tree if and only if $\sum_i d_i = 2(n-1)$ [19]. Since this condition can be verified in NCC$0$ in $O(\log n)$ rounds by aggregation, we can quickly test whether a given degree sequence has a valid tree-realization.

We present two $O(\text{polylog}(n))$ round algorithms for realizing trees in NCC$0$. The first is a simple algorithm yielding a tree-realization of the maximum possible diameter. We then show how with some modification to the first algorithm, it is possible to output a minimum diameter tree.

A detailed pseudocode of our first algorithm is presented in Algorithm 4. Given a realizable sequence, the key idea here is as follows. First we sort the nodes according to their degrees. This allows us to separate out the leaves and the non-leaves. Let $V = \{x_1, x_2, \ldots, x_n\}$ be the nodes in the network such that $d(x_1) \geq d(x_2) \geq \ldots \geq d(x_n)$. Assume that they are arranged in a sorted path graph $L$. Define $k = |\{x \mid d(x) > 1\}|$. Thus, $x_{k+1}, \ldots, x_n$ are the leaves. First, for each $i$ in $[1, k]$, $x_i$ creates an edge with $x_{i+1}$ by exchanging IDs. Now for any $x_i$ such that $i \in [2, k]$, as that node has a predecessor and a successor, the remaining degree requirement is $d(x_i) - 2$; for $x_1$, the degree requirement is $d(x_1) - 1$. Each of these can be satisfied by simply attaching the required number of leaves (as the analysis in [1] shows that there will be a sufficient number of leaves). Once $k$ is made known to the nodes, node $x_i$ is only required to know how many leaves are required for the nodes $x_1, \ldots, x_{i-1}$. In the algorithm, we capture this by the prefix sum $p_i$, which allows node $x_i$ to calculate the number of children its predecessors require. Once $p_i$ is known, $x_i$ knows the position of its leaves in $L$ and can inform its ID to them.

We now show how to achieve a polylogarithmic implementation of Algorithm 4. Steps 1-3 can be performed by a combination of sorting and aggregation. First, sort the nodes according to their degrees, after which each node can
Algorithm 4: Distributed-Tree-Realization-1

Input: An $n$-node network with each node $x$ provided with a degree $d(x)$.
Output: A corresponding implicit tree realization.

1. Sort the nodes into a list $L$, represented as $x_1, \ldots, x_n$, in non-increasing order of degrees. After sorting, each node knows its position in $L$.
2. Aggregate the value $S = \sum_{x \in L} d(x)$ at a single node (say $x_1$). If $S \neq 2(n - 2)$, then $x_1$ broadcasts UNREALIZABLE, and the procedure terminates.
3. Aggregate and broadcast to all the nodes the value $k = \{x \mid d(x) > 1\}$ such that $x_1, \ldots, x_k$ are the non-leaf nodes and $x_{k+1}, \ldots, x_n$ are leaves in $L$.
4. Compute the prefix sums $p_i = 2 + \sum_{j=1}^{i-1} (d_j - 2)$ for $2 \leq i \leq k$. $p_1 = 2$.

5. foreach $i \in\{k\}$, in parallel do
   6. if $i = 1$ then
      7. Set $I = 0$
   8. else Set $I = 1$;
   9. Node $x_i$ stores ID($x_{i-1}$) and ID($x_{i+1}$) in its neighbor-list (when exists).
10. Broadcast ID($x_i$) to the following $d_i - 1 - I$ contiguous leaves $x_{k+p_i+I}, \ldots, x_{k+p_i+d_i-2}$.
11. The nodes $x_{k+p_i+I}, \ldots, x_{k+p_i+d_i-2}$ store ID($x_i$) in their neighbor-list.
12. %there will always be sufficiently many leaves ([11], section 3.2)

be made to know its position in the path (via the BFS tree construction). Then we perform steps 2 and 3 by global aggregation operations (to the node in position 1, $x_1$).

The prefix sums $p_i$ (in step 4) can be computed in $O(\log n)$ rounds in a manner that is reminiscent of computing in-order traversal numbers. In fact, in-order traversal numbers are (close to) prefix sums when the degree values are 1. Our approach is to build a local binary tree on the nodes $x_1, \ldots, x_k$. Then we apply a two-phase process. Phase 1 involves a bottom-up convergecast (to calculate sum of the degree values of nodes in sub-trees). At the end of phase 1, the root will know its prefix sum and phase 2 involves a recursive top-down computation in the binary tree for nodes to deduce their prefix sums.

Step 9 can be performed in $O(1)$ rounds by neighbors simply exchanging their IDs. Note that at the end of step 4, each node $x_i$ (for any $i \in\{1, k\}$) knows how many leaves it requires. Thus any $x_i$ can locally calculate the positions of its leaves in the path $L$. Suppose $x_i$’s leaves are located in positions $j$ to $j + (d_i - 3)$. Then $x_i$ can inform these nodes of its ID in the following manner. First, using $j$ as the group ID, $x_i$ informs $x_j$ of its ID using the aggregation algorithm. Then it does the same for node $x_{j+d_i-3}$. Note that this would only require $O(\log n)$ rounds via aggregation (Theorem 6). We now treat the problem of the nodes $x_j$ and $x_{j+d_i-3}$ informing the other nodes between their position in $L$ as a smaller instance of the global broadcast problem in the NCC (containing only the $d_i - 2$ nodes), which can be solved in $O(\log n)$ rounds (Theorem 4). Also, each $x_i$ can simultaneously inform its ID to all its leaves without congestion by running the above procedure concurrently. Thus we have,

**Theorem 14.** There exists a procedure for implicitly realizing a length-$n$ tree realizable sequence by a tree overlay network in $O(\log^3 n)$ rounds in the NCC$_0$ model.

Next, we give an algorithm for obtaining a minimum diameter tree realization from a degree sequence. The resulting Algorithm 5 is in fact a distributed version of the sequential algorithm studied in [30], and the resulting tree is referred to therein as the greedy tree $T_G$.

The key idea here is to put the nodes with the higher degrees as high up in the tree as possible, without violating the realization requirement. Initially, the highest degree node $x_1$ becomes the root and connects to next $d(x_1)$ highest degree nodes. Thereafter, each node $x_i$ satisfies its degree requirement by connecting with $d(x_i) - 1$ nodes (as it is already connected to its parent) of highest degree that do not yet have a parent.

We next claim that the greedy tree $T_G$ constructed by Algorithm 5 has minimum diameter.

**Lemma 15.** The output of Algorithm 5 is a minimum diameter realization of the input degree sequence $D$.

**Proof.** Let $T$ be the class of trees realizing $D$. The eccentricity of a node $v$ in the tree $T$ is $ecc(v, T) = \max_{u \in T} dist_T(v, u)$. Let $n_\ell(T) = |\{v \mid ecc(v, T) \leq \ell\}|$. It is shown in [30] (Theorem 16) that $n_\ell(T_G) \geq n_\ell(T)$ for every non-negative integer $\ell$ and tree $T \in T$. Fix $\ell = \delta_G - 1$, where $\delta_G$ is the diameter of $T_G$. Then
Algorithm 5: Distributed-Tree-Realization-2

**Input:** An \( n \)-node network with each node \( x \) provided its degree \( d(x) \) in the tree.

**Output:** An implicit tree realization of min diameter.

1. **Initialization:** As in Algorithm 4, line numbers 1 – 3.
2. Compute the prefix sums \( p_i = 2 + \sum_{j=1}^{i-1} (d_j - 1) \) for \( 2 \leq i \leq k \). Set \( p_1 = 2 \).
3. \( \textbf{foreach } i \in [n], \text{ in parallel do} \)
   
   \( \quad \text{if } i = 1 \text{ then} \)
   
   \( \quad \quad \text{Set } I = 0 \)
   
   \( \quad \text{else Set } I = 1; \)
   
   \( \quad \text{Broadcast } ID(x_i) \text{ to the } d_i - 1 \text{ nodes } x_{p_i+1}, \ldots, x_{p_i+d_i-1}. \)
   
   \( \quad \text{The nodes } x_{p_i+1}, \ldots, x_{p_i+d_i-1} \text{ store } ID(x_i) \text{ in their neighbor-list.} \)

\( n_\ell(T_G) \leq n - 2 \). Since \( T_G \) maximizes \( n_\ell(T) \), also \( n_\ell(T') \leq n - 2 \) for every other \( T' \in \mathcal{T} \). Hence, every \( T' \in \mathcal{T} \) must have diameter at least \( \ell + 1 = \delta_G \), establishing the lemma.

In the NCC model, Steps 1-2 of Algorithm 5 can be performed in \( \text{polylogarithmic} \) rounds via sorting and global aggregation. Step 3 is performed just as step 4 in Algorithm 4. Steps 8-10 can be performed in a similar manner. Each node \( x_i \), once it knows its children’s positions in the path (via the prefix sums), does the following. Suppose \( x_i \)'s children are in positions \( j \) to \( j + d_i - 2 \). Then \( x_i \) informs node \( x_j \) and \( x_{j+d_i-2} \) of its ID (by a multicast). Then these two children inform the other nodes (as before this can be seen as a smaller instance of broadcast in the NCC with \( d_i - 1 \) nodes). Note that no node will belong to more than one multicast group. Thus, we get:

**Theorem 16.** There exists a procedure for implicitly realizing a length-\( n \) tree realizable sequence \( D \) by an overlay tree network \( T \) in \( O(\text{polylog}(n)) \) rounds, in the NCC model, such that \( T \) has the minimum diameter possible for the given \( D \).

### 6 Connectivity Threshold Realizations

In this section we study the minimum connectivity threshold realization problem in the NCC model. The connectivity between nodes \( u \) and \( v \) in graph \( G \) is given by \( \text{Conn}_G(u, v) \), that refers to the minimum number of edge disjoint paths between \( u \) and \( v \). Initially, each node \( v \) in the network is locally provided with a connectivity threshold vector \( \sigma(v) = (\sigma(v, u_1), \ldots, \sigma(v, u_n)) \), which specifies the required minimum edge connectivity that needs to be satisfied at \( v \) w.r.t. every other node \( u \neq v \). Normally, the goal is then to compute the sparsest possible graph \( G \) with, say \( m \) edges, such that any two nodes \( u, v \) in \( G \) satisfy \( \text{Conn}_G(u, v) \geq \sigma(u, v) \).

Here, we focus on a 2-approximate solution, ensuring that the number of edges in the overlay network is at most \( 2m \).

To do this, we actually require a stronger condition. Define \( \rho(v) = \max_{u \in V} \sigma(u, v) \) for each \( v \in V \). Our algorithms output realizations that satisfy the following condition:

\[
\text{Conn}_G(u, v) \geq \min\{\rho(u), \rho(v)\}.
\]

This allows us to assume that each node \( v \) is provided with just the value \( \rho(v) \), rather than the length-\( n \) vector \( \sigma(v) \), or equivalently, that all the entries of \( \sigma(v) \) are identical to \( \rho(v) \).

#### 6.1 An \( \tilde{O}(1) \) Time Implicit Realization in NCC\(_1\)

We first consider the simplest scenario of implicit realization in the NCC\(_1\) model. Our algorithm has two steps.

1. We first find a node \( w \) of maximum \( \rho \) value, i.e., such that \( \rho(w) = \max_{v \in V} \rho(v) \), breaking ties arbitrarily. The node \( w \) is found using data-aggregation, and its address is broadcast to all the nodes, all in \( \tilde{O}(1) \) time.
2. Next, each node \( v \neq w \) chooses an arbitrary subset \( X^v = \{x_{\rho(v)}^1, \ldots, x_{\rho(v)}^\rho\} \) of \( V \) satisfying that \( w \in X^v \), and outputs \( X^v \times \{v\} \) as the edges stored at node \( v \). This step is done in \( O(1) \) time in the NCC\(_1\)-model since \( v \) already knows the addresses of all the nodes in \( X^v \).
Correctness: Consider any node \( v \neq w \). We assume w.l.o.g. that \( x_{v}^{i} \), the first node in \( X^{v} \), be \( w \). Then \((v, w)\), and \((v, x_{v}^{i})\) for \( i \in [2, \rho(v)]\), are \( \rho(v) \) edge disjoint paths from \( v \) to \( w \), proving that \( \text{Conn}_{G}(w, v) = \rho(v) \) for every \( v \neq w \). Now for any two nodes \( v_{1}, v_{2} \neq w \), by Menger’s Theorem [17] \( \text{Conn}_{G}(v_{1}, v_{2}) \geq \min\{\text{Conn}_{G}(v_{1}, w), \text{Conn}_{G}(v_{2}, w)\} = \min\{\rho(v_{1}), \rho(v_{2})\} \). Node \( w \) satisfies the connectivity requirement, since it has edges to all the other nodes.

Approximation factor: Observe that any graph that adheres to the minimum connectivity threshold constraints, the degree of each node \( v \) must be at least \( \rho(v) \). Thus, any such realizing graph must contain at least \( \frac{1}{2} \sum_{v \in V} \rho(v) \) edges. Our algorithm obtains a realization \( G \) where each node \( v \neq w \) adds \( \rho(v) \) edges. (Since each node \( v \neq w \) adds an edge to \( w \), \( w \) need not add any more edges.) The total number of edges is determined by \( \sum_{v \in V \setminus \{w\}} \rho(v) \leq \sum_{v \in V} \rho(v) \) edges. Hence, our algorithm gives a 2-approximation solution.

\[ \text{Theorem 17. There exists a } O(1) \text{ time procedure for implicitly realizing, in the NCC}_{1} \text{ model, a given collection of connectivity threshold vectors by a graph } G \text{ such that number of edges in } G \text{ is at most twice the number of edges in the optimal realization.} \]

6.2 An \( O(\Delta) \) Time Explicit Realization in \( \text{NCC}_{0} \) and \( \text{NCC}_{1} \)

In this section, we present a connectivity threshold algorithm that works also in the \( \text{NCC}_{0} \) model. The algorithm crucially uses the degree-realization results of Section 4. Our results here are inspired by the work of Frank and Chou [15] that presents a 2-approximation in the centralized setting.

The algorithm begins by first sorting the nodes in the non-increasing order of \( \rho \). Let \( x_{1}, \ldots, x_{n} \) be the sorted nodes and let \( d_{0} = \rho(x_{1}) \). The algorithm proceeds in two phases.

1. The first phase focuses on the nodes \( x_{1}, \ldots, x_{d_{0}+1} \). The node \( x_{1} \) broadcasts \( d_{0} \) to everyone. Next, the nodes \( x_{1}, \ldots, x_{d_{0}+1} \) try to realize \( (\rho(x_{1}), \rho(x_{2}), \ldots, \rho(x_{d_{0}+1})) \) as a degree sequence (i.e., using the \( \rho \) values as assigned degrees) either exactly or approximately using the algorithm given in Section 4.3. This ensures that each node \( x_{i} \), for \( i \in [2, d_{0} + 1] \), in the partially computed graph is connected to \( x_{1} \) and to at least \( \rho(x_{i}) - 1 \) nodes from the set \( x_{2}, \ldots, x_{d_{0}+1} \).

2. In the second phase, for each \( i \in [d_{0} + 2, n] \), in parallel, \( x_{i} \) broadcasts \( ID(x_{i}) \) to \( \rho(x_{i}) \) predecessors of \( x_{i} \), i.e., \( x_{i-1}, x_{i-2}, \ldots, x_{1} \). These nodes store \( ID(x_{i}) \) in their neighbour-list. This ensures that for every \( i \in [d_{0} + 2, n] \), the degree of \( x_{i} \) in the graph induced by the nodes \( x_{1}, \ldots, x_{i} \) is at least \( \rho(x_{i}) \). To make the realization explicit, the nodes \( x_{i-1}, \ldots, x_{1} \) also broadcast their own IDs to \( x_{i} \) (after receiving \( x_{i} \)’s ID).

\[ \text{Algorithm 6: Distributed-Connectivity-Realization} \]

**Input:** An \( n \)-node network with each node \( x \) provided with a connectivity threshold \( \rho(x) \).

**Output:** An explicit connectivity threshold realization.

1. Sort the nodes in the non-increasing order of \( \rho \), let these be represented as \( x_{1}, \ldots, x_{n} \). (After sorting, each node knows its position in the sorted list).
2. Broadcast \( d_{0} = \rho(x_{1}) \) to all the nodes.
3. Obtain a distributed degree-realization for sequence \( (\rho(x_{1}), \rho(x_{2}), \ldots, \rho(x_{d_{0}+1})) \) over the first \( d_{0} + 1 \) nodes using Theorem 13.
4. **foreach** \( i \in [d_{0} + 2, n] \), in parallel, do
   5. Broadcast address \( (x_{i}) \) to \( \rho(x_{i}) \) predecessors of \( x_{i}, x_{i-1}, x_{i-2}, \ldots, x_{1} \).
   6. The nodes \( x_{i-1}, \ldots, x_{1} \) after receiving the address of \( x_{i} \) store it in their neighbor-list, and also broadcast their own addresses to node \( x_{i} \), which in turn stores them in its neighbor-list.

**Correctness:** Let \( G_{1} \) be the graph induced by the nodes \( x_{1}, \ldots, x_{d_{0}+1} \) computed in the first phase, and \( G_{2} = G \) be final graph after the second phase. We show that \( \text{Conn}_{G}(x_{1}, x_{i}) \geq \rho(x_{i}) \), for each \( i > 1 \). First consider \( G_{1} \). Since degree of \( x_{1} \) in \( G_{1} \) is \( d_{0} \), it is adjacent to all nodes in \( G_{1} \). Consider a node \( x_{i} \), for \( i \in [2, d_{0} + 1] \). Let \( u_{1}, \ldots, u_{\rho(x_{i})-1} \) be \( x_{i} \)'s neighbors in \( G_{1} \), other than \( x_{1} \). Then \( (x_{1}, x_{i}) \) and \( (x_{i}, u_{j}, x_{1}) \) for \( j \in [1, \rho(x_{i}) - 1] \) are \( \rho(x_{i}) \) edge disjoint paths from \( x_{1} \) to \( x_{i} \). Since \( G_{1} \) is a subgraph of \( G \), for each \( i \in [2, d_{0} + 1] \), \( \text{Conn}_{G}(x_{1}, x_{i}) \geq \text{Conn}_{G_{1}}(x_{1}, x_{i}) \geq \rho(x_{i}) \). It is easy to prove by induction that for \( i \geq d_{0} + 2 \), \( \text{Conn}_{G}(x_{1}, x_{i}) \geq \rho(x_{i}) \). Finally, using Menger’s Theorem [17], we get \( \text{Conn}_{G}(x_{1}, x_{j}) \geq \min\{\text{Conn}_{G}(x_{1}, x_{i}), \text{Conn}_{G}(x_{1}, x_{j})\} \geq \min\{\rho(x_{i}), \rho(x_{j})\} \).

**Approximation factor:** By Theorem 13, the number of edges in \( G_{1} \) is at most \( \sum_{i=1}^{d_{0}+1} \rho(x_{i}) \). The second phase adds \( \rho(x_{i}) \) edges for every \( i \geq d_{0} + 2 \). Hence the number of edges in \( G \) is at most \( \sum_{i=1}^{n} \rho(x_{i}) \). By arguments similar
to those of Subsection 6.1. $\frac{1}{2} \sum_{i=1}^{n} \rho(x_i)$ is a lower bound on the number of edges in $G$, implying that our algorithm achieves an approximation factor of two. We thus have the following.

**Theorem 18.** There exists a $\tilde{O}(\Delta)$ time procedure for explicitly (as well as implicitly) realizing, in the NCC$_0$ and NCC$_1$ models, a connectivity threshold realizing graph $G$ such that number of edges in $G$ is at most twice the number of edges in the optimal realization.

### 7 Lower Bounds for Degree Realization

In this section, we establish a number of lower bounds applicable to the distributed graph realization problem in NCC$_0$, some of which are fairly straightforward. We begin with the simplest case.

In any instance of the explicit version, a node of maximum degree needs to know the addresses of its $\Delta$ neighbors. Thus,

**Theorem 19.** Any distributed NCC$_0$ algorithm for explicit realization of a degree sequence $D$ with maximum degree $\Delta$ requires at least $\Omega(\Delta / \log n)$ rounds for all instances.

We now turn our attention to implicit realizations. Let $D_{n,m}$ be the class of length-$n$ degree sequences $D = (d_i)_{i \in [n]}$ such that $m = \sum_i d_i / 2$. Furthermore, let $D'_{\Delta}$ be the class of length-$n$ degree sequences with maximum degree $\Delta$.

**Theorem 20.** Any distributed algorithm to realize degree sequences in $D_{n,m}$ in NCC$_0$ requires at least $\Omega(\sqrt{m} / \log n)$ rounds. Similarly, any distributed algorithm to realize degree sequences in $D'_{\Delta}$ in NCC$_0$ requires at least $\Omega(\Delta)$ rounds.

**Proof.** Let $k = \lceil \sqrt{m} \rceil$, and let $D'_{n,m} \subset D_{n,m}$ be the family of degree sequences in which $d_i = 0$ for all $i > k$.

Consider an arbitrary $D^* = (d_1, d_2, \ldots, d_k, d_{k+1}, \ldots, d_n) \in D'_{n,m}$. Any NCC$_0$ algorithm must ensure that the first $k$ nodes combined learn $\Omega(m)$ IDs. By the pigeonhole principle, at least one of these nodes must learn $\Omega(m/k) = \Omega(\sqrt{m})$ IDs, which requires $\Omega(\sqrt{m} / \log n)$ rounds in NCC$_0$.

The lower bound of $\Omega(\Delta)$ for realizing degree sequences in $D'_{\Delta}$ can be argued similarly by considering the degree sequence $(d_i = \Delta)_{1 \leq i \leq n}$. \qed

### 8 Conclusion and Future Work

We have initiated the study of graph realization problems in the distributed setting, and presented efficient algorithms in the NCC model for realizing overlay networks that satisfy degree or connectivity requirements. We believe that such formal study of overlay network design may facilitate the design of a wide range of fundamental tools with strong guarantees. In addition to building such theoretical foundation, we hope our line of work may produce new ideas leading to practical improvements in building overlay networks. Our work also opens up a number of interesting new directions for future exploration.

1. First, it is unclear if the lower bounds we have provided for NCC$_0$ hold also for NCC$_1$. If not, it will be interesting to achieve improved algorithms for NCC$_1$.
2. Overlays need to be robust against a wide range of failures. Modeling these requirements in the form of properties that must be satisfied could lead to new graph realization variants that have not been studied so far.
3. It will be interesting to realize graphs from suitable graph classes. We have already studied realization of trees, but we believe that realizing graphs in other classes like planar graphs, chordal graphs, etc. can also be of value.

**Acknowledgments**

John Augustine and Sumathi Sivasubramaniam are supported by DST/SERB Extra Mural Grant (file number EMR/2016/00301). John Augustine is also supported by DST/SERB MATRICS Grant (file number MTR/2018/001198). David Peleg is supported by the Venky Harinarayanan and Anand Rajaraman Visiting Chair Professorship at IIT Madras. Supported by the chair’s funds, the work was done in part when David Peleg, Avi Cohen, and Keerti Choudhary visited IIT Madras and when John Augustine visited Weizmann Institute of Science. This work was done in part when Suman Sourav visited IIT Madras. Finally, we note that a short version of this paper will appear in the proceedings of IPDPS 2020.
References

[1] S.R. Arikati and A. Maheshwari. Realizing degree sequences in parallel. *SIAM J. Discr. Math.*, 9:317–338, 1996.
[2] J. Aspnes and G. Shah. Skip graphs. *Tr. Algo.*, 3, 2007.
[3] J. Augustine, M. Ghaffari, R. Gmyr, K. Hinnenthal, C. Scheideler, F. Kuhn, and J. Li. Distributed computation in node-capacitated networks. *SPAA*, 69–79, 2019.
[4] J. Augustine and S. Sivasubramaniam. Spartan: A framework for sparse robust addressable networks. *IPDPS*, 1060–1069, 2018.
[5] B. Awerbuch and C. Scheideler. The hyperring: a low-congestion deterministic data structure for distributed environments. *15th SODA*, 318–327, 2004.
[6] A. Bar-Noy, K. Choudhary, D. Peleg, and D. Rawitz. Realizability of graph specifications: Characterizations and algorithms. *25th SIROCCO*, 3–13, 2018.
[7] A. Bar-Noy, K. Choudhary, D. Peleg, and D. Rawitz. Efficiently realizing interval sequences. *ISAAC (to appear)*, 2019.
[8] A. Bar-Noy, K. Choudhary, D. Peleg, and D. Rawitz. Graph profile realizations and applications to social networks. *13th WALCOM*, 1–12, 2019.
[9] M. Behzad and J.E. Simpson. Eccentric sequences and eccentric sets in graphs. *Discr. Math.*, 16:187–193, 1976.
[10] P. Erdös and T. Gallai. Graphs with prescribed degrees of vertices. *Matematikai Lapok*, 11:264–274, 1960.
[11] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Acad. Sci. Imperialis Petropol.*, 8:128–140, 1736.
[12] A. Fiat and J. Saia. Censorship resistant peer-to-peer content addressable networks. *SODA*, 94–103, 2002.
[13] A. Frank. Augmenting graphs to meet edge-connectivity requirements. *SIAM J. Discr. Math.*, 5:25–43, 1992.
[14] A. Frank. Connectivity augmentation problems in network design. *Mathematical Programming: State of the Art*, 34–63, 1994.
[15] H. Frank and W. Chou. Connectivity considerations in the design of survivable networks. *IEEE Tr. Circuit Theory*, CT-17:486–490, 1970.
[16] R.E. Gomory and T.C. Hu. Multi-terminal network flows. *J. Soc. Industrial & Applied Math.*, 9, 1961.
[17] Frank Göring. Short proof of Menger’s theorem. *Discrete Mathematics*, 219(1-3):295–296, 2000.
[18] S. L. Hakimi. On realizability of a set of integers as degrees of the vertices of a linear graph – I. *SIAM J. Appl. Math.*, 10:496–506, 1962.
[19] F. Harary. *Graph Theory*. Addison-Wesley, 1971.
[20] V. Havel. A remark on the existence of finite graphs. *Casopis Pest. Mat.*, 80:477–480, 1955.
[21] P. Hell and D. G. Kirkpatrick. Linear-time certifying algorithms for near-graphical sequences. *Discr. Math.*, 309:5703–5713, 2009.
[22] J. H. Korhonen and J. Suomela. Towards a complexity theory for the congested clique. *SPAA*, 163–172, 2018.
[23] L. Lesniak. Eccentric sequences in graphs. *Periodica Math. Hungarica*, 6:287–293, 1975.
[24] Z. Lotker, B. Patt-Shamir, E. Pavlov, and D. Peleg. Minimum-weight spanning tree construction in $o(\log \log n)$ communication rounds. *SIAM J. Computing*, 35:120–131, 2005.
[25] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications & Tutorials*, 7:72–93, 2005.
[26] A. Malatras. State-of-the-art survey on P2P overlay networks in pervasive computing environments. *J. Network & Computer Applic.*, 55:1–23, 2015.
[27] M. Mitzenmacher and E. Upfal. *Probability and Computing*. Cambridge Univ. Press, 2017.
[28] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, USA, 2000.
[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *SIGCOMM. ACM*, 2001.
[30] H. C. Smith, L. A. Székely, and H. Wang. Eccentricity sums in trees. *Discrete Applied Mathematics*, 120–131, 2016.
[31] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE Tr. Networking*, 11:17–32, 2003.

[32] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec. Peer-assisted content distribution in akamai netsession. *IMC*, 31–42. ACM, 2013.