Algorithmica (2024) 86:2959–2996
https://doi.org/10.1007/s00453-024-01250-3

Sublinear Algorithms in T-Interval Dynamic Networks

Irvan Jahja[1] · Haifeng Yu[1]

Received: 12 April 2022 / Accepted: 22 June 2024 / Published online: 12 July 2024
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

Abstract

We consider standard T*-interval dynamic networks*, under the synchronous timing model and the broadcast CONGEST model. In a T*-interval dynamic network*, the set of nodes is always fixed and there are no node failures. The edges in the network are always undirected, but the set of edges in the topology may change arbitrarily from round to round, as determined by some *adversary* and subject to the following constraint: For every T consecutive rounds, the topologies in those rounds must contain a common connected spanning subgraph. Let H_r to be the maximum (in terms of number of edges) such subgraph for round r through $r + T - 1$. We define the *backbone diameter* d of a T-interval dynamic network to be the maximum diameter of all such H_r's, for $r \geq 1$. We use n to denote the number of nodes in the network. Within such a context, we consider a range of fundamental distributed computing problems including COUNT/MAX/MEDIAN/SUM/LEADERELECT/CONSENSUS/CONFIRMEDFLOOD. Existing algorithms for these problems all have time complexity of $\Omega(n)$ rounds, even for $T = \infty$ and even when d is as small as $O(1)$. This paper presents a novel approach/framework, based on the idea of *massively parallel aggregation*. Following this approach, we develop a novel deterministic COUNT algorithm with $O(d^3 \log^2 n)$ complexity, for T-interval dynamic networks with $T \geq c \cdot d^2 \log^2 n$. Here c is a (sufficiently large) constant independent of d, n, and T. To our knowledge, our algorithm is the very first such algorithm whose complexity does not contain a $\Theta(n)$ term. This paper further develops novel algorithms for solving MAX/MEDIAN/SUM/LEADERELECT/CONSENSUS/CONFIRMEDFLOOD, while incurring

The authors of this paper are alphabetically ordered. A preliminary conference version [1] of this work was published in ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), July 2020. That preliminary conference version does not contain any of the proofs in this (journal) paper. These proofs constitute an integral and central component of the research contributions. In addition, this (journal) paper has also incorporated various other improvements, such as updated discussions on related works.

✉ Haifeng Yu
haifeng@comp.nus.edu.sg

Irvan Jahja
irvan@comp.nus.edu.sg

1 National University of Singapore, Singapore, Republic of Singapore

$O(d^3\text{polylog}(n))$ complexity. Again, for all these problems, our algorithms are the first ones whose time complexity does not contain a $\Theta(n)$ term.

Keywords Distributed algorithms · T-interval dynamic networks · Sublinear algorithms

1 Introduction

1.1 Our Setting

We consider various fundamental distributed computing problems in standard T-*interval dynamic networks* [2–7], under the synchronous timing model. The network has a fixed set of n nodes (i.e., no node addition/removal), which proceed in lock-step *rounds*, starting from round 1. Each node has a unique *id* of $O(\log n)$ size. The algorithm knows neither n nor any upper bound on n.[1] In a T-*interval dynamic network* ($T \geq 1$), the edges are always undirected, but the set of edges in the topology may change arbitrarily from round to round, as determined by some *adversary* and subject to the following constraint: For every T consecutive rounds, the topologies in those rounds must contain a common connected spanning subgraph, which implies that this subgraph remains stable in those T rounds. (Note that this subgraph is required to be both connected and spanning.) Let H_r to be the maximum (in terms of number of edges) such subgraph for the T rounds from round r to $r + T - 1$. We define the *backbone diameter* d of a T-interval dynamic network to be the maximum diameter of all such H_r's, for $r \geq 1$. The distributed algorithm knows neither d nor any upper bound on d. The above notions can also be extended to $T = \infty$. Namely, in an ∞-*interval dynamic network*, the adversary guarantees that the topologies in all rounds contain a common connected spanning subgraph. Let H be the maximum (in terms of number of edges) such subgraph. We define the *backbone diameter* d of an ∞-interval dynamic network to be the diameter of this graph H. This paper will focus on T-interval dynamic networks with sufficiently large T (see exact condition later), including $T = \infty$.

Following [2, 3, 6], we use the broadcast CONGEST model [8] where in each round, each node is allowed to choose a single message of $O(\log n)$ size, and send the message simultaneously to all its neighbors. (A node cannot send different messages to different neighbors.) Without loss of generality, we assume that a message always contains its sender's id. At the end of each round, each node receives all the messages sent in that round by all its neighbors (as determined by the topology of that round). Note that, a node does not know its neighbors, before it receives messages from them. Also, a node does not know the topology in each round.

The *time complexity* (or simply *complexity*) of a distributed algorithm for solving a certain problem is defined to be the number of rounds needed for all nodes to output

[1] As in [2, 3, 5, 6], we have assumed that each node has a unique id of size $O(\log n)$. This means the largest id among the n nodes maps to a loose polynomial upper bound on n. However, finding the largest id among the n nodes is at least as hard as the LEADERELECT problem (formally defined later), and hence is non-trivial by itself.

and terminate, under the worst-case input and worst-case adversary.[2] We describe the time complexity as a function of n and d. The central challenge in designing distributed algorithms in T-interval dynamic networks is that the H_r's (or H) and d are all unknown to the algorithm, and that in each round, the algorithm does not know beforehand which edges in the network will survive and which edges will be deleted/added.

1.2 Problems and Existing Results

We consider the following fundamental distributed computing problems in T-interval dynamic networks, where each node has an *input* of $O(\log n)$ size (for some problems, the input is simply a dummy value):

- COUNT: All nodes should output n.
- MAX/MEDIAN/SUM: All nodes should output the max/median/sum of the n inputs (as integers).
- LEADERELECT: A unique leader should be elected, and all nodes should output the leader's id.
- CONSENSUS: All nodes should output some common consensus value, while satisfying the standard agreement (i.e., all nodes' decisions are the same), validity (i.e., decision should equal input if all inputs are identical), and termination (i.e., all nodes eventually decide) requirements [9].
- CONFIRMEDFLOOD [10]: One distinguished node needs to propagate its input to all nodes, and then should output "1" *after* all nodes have received its input.

We note that all these distributed computing problems are non-trivial to solve, even in ∞-interval dynamic networks, given that d and H are not known beforehand.[3] To our knowledge, all existing deterministic and randomized[4] algorithms [2, 4–7] for all the above problems have time complexity of $\Omega(n)$ rounds in T-interval dynamic networks,[5] even for $T = \infty$. A dynamic network's backbone diameter d may range from 1 to $n - 1$. While the existing works [2, 4–7] do not describe their algorithms' time complexities in terms of both n and d, it can be easily verified that their time complexities remain $\Omega(n)$ even when d is as small as $O(1)$ (and even when $T = \infty$).

[2] We will mainly be concerned with deterministic algorithms, where it is irrelevant whether the adversary can see and then adapt to the coin flip outcomes in the algorithm. (Namely, it is irrelevant whether the adversary is *oblivious* or *adaptive*.)

[3] If H is known, then regardless of whether d is known, one can trivially solve all these problems in $O(d)$ rounds, by doing simple tree-based aggregation over edges in H. If H is not known but d is known, then MAX/LEADERELECT/CONSENSUS/CONFIRMEDFLOOD can all be solved trivially via flooding in $O(d)$ rounds, while COUNT/MEDIAN/SUM remain non-trivial. If neither H nor d is known, then all these problems are non-trivial.

[4] A randomized algorithm is designed either for *oblivious adversaries* or *adaptive adversaries*. The complexity of a randomized algorithm is always defined under the worst-case adversary for which the algorithm is designed.

[5] In addition to these algorithms [2, 4–7] designed for our setting, researchers have also developed algorithms for solving the above problems in *anonymous* dynamic networks (e.g., [11–16]). Obviously, the anonymous setting is harder, and those algorithms for anonymous dynamic networks also all have $\Omega(n)$ complexity. See more discussion on anonymous dynamic networks later.

Putting it another way, even if we describe these complexities as functions of both n and d, such functions would still all be $\Omega(n)$. Part of the reason for such $\Omega(n)$ complexity is that existing approaches often solve these problems by each node collecting all n inputs—namely, these problems are often solved as a byproduct of solving *token dissemination* [2, 4–6]. But token dissemination fundamentally requires each node to receive $\Omega(n \log n)$ bits, which takes $\Omega(n)$ rounds for a constant degree node, even when $d = O(1)$.

Solving Count/Max/Median/Sum/LeaderElect/Consensus/Con- firmed Flood by collecting all n inputs obviously appears to be quite an overkill. All these problems are "global" in the sense that the output could be affected by far away nodes in the network. Such a need for "global" information does lead to an $\Omega(d)$ lower bound, but not an $\Omega(n)$ lower bound. While there is no hope of getting $o(n)$ complexity when $d = \Theta(n)$, it seems that we still should be able to solve these "global" problems in less than n rounds when d is small. Despite such natural thoughts, no existing algorithms can achieve this.

1.3 Our Results

This paper presents a novel approach/framework, based on the idea of *massively parallel aggregation*, for designing algorithms in T-interval dynamic networks. Following our approach/framework, we develop a novel deterministic algorithm for solving Count with $O(d^3 \log^2 n)$ complexity, for T-interval dynamic networks with $T \geq c \cdot d^2 \log^2 n$. (Throughout this paper, whenever we say T is larger than some value, it always includes the limiting case of $T = \infty$.) Here c is a (sufficiently large) constant independent of d, n, and T. To our knowledge, our algorithm is the very first such algorithm whose complexity does not contain a $\Theta(n)$ term. In fact, even for the easier case of $T = \infty$, existing algorithms still have $\Omega(n)$ time complexity.

For $d = O(n^a)$ with constant $a < \frac{1}{3}$, our deterministic Count algorithm has $o(n)$ complexity, which is better than all existing (both deterministic and randomized) Count algorithms in this setting. For $d = O(\text{polylog}(n))$, our algorithm is exponentially faster than all existing algorithms.

By applying our approach/framework to other problems, the paper further develops novel algorithms for solving Max/Median/Sum/LeaderElect/Consensus/Confir medFlood, while incurring either $O(d^3 \log^2 n)$ or $O(d^3 \log^3 n)$ complexity, in T-interval dynamic networks with $T \geq cd^2 \log^2 n$ for some (sufficiently large) constant c. Again, for all these problems, our algorithms are the first ones without a $\Theta(n)$ term in its complexity, achieving $o(n)$ complexity when $d = O(n^a)$ with constant $a < \frac{1}{3}$.

1.4 Further Implications of Our Results

It is known that even when $d = O(1)$, problems such as Count and Sum have $\Omega(\text{poly}(n))$ lower bounds in all the following settings:

- Same as our ∞-interval setting except that in each round, a node can choose to either send a message or receive messages, but cannot do both [10].[6]
- Same as our ∞-interval setting except that the set of nodes (instead of the edges) can change (by crashing) from round to round [17].
- Same as our ∞-interval setting except that the topology is *directed* and never changes [18].

Despite some of the above settings being seemingly close to our setting, our $O(d^3 \log^2 n)$ upper bound for COUNT and SUM implies that such $\Omega(\text{poly}(n))$ lower bound can never carry over to our setting. The only currently known lower bound in our setting is the trivial $\Omega(d)$ lower bound.

1.5 Discussion on Our Approach/Framework

Our novel approach/framework is quite different from existing algorithmic techniques in T-interval dynamic networks. At the highest level, our approach starts from the classic idea of tree-based aggregation. Let us take COUNT as an example. In this classic approach, there is a rooted spanning tree and each node contributes a value of 1. These values are propagated upstream along the tree paths to the root, while being aggregated (i.e., summed together) along the way. The root can then eventually learn the total count of nodes from the final sum. In dynamic networks however, such aggregation can be easily disrupted by the topology changes. To deal with this, conceptually, we do *massively parallel* aggregation simultaneously along many (up to exponential number of) aggregation paths from each node to the tree root. Next, we *stagger* the aggregation, together with carefully designed *re-tries*, to limit the adversary's damage. Finally, we use a number of tricks (e.g., by allowing non-distinct nodes in an aggregation path), to limit the amount of bookkeeping needed when dealing with a large number of aggregation paths—otherwise the design would have been highly inefficient.

1.6 More Discussion on Related Works

As mentioned earlier, in the T-interval model, researchers often solve problems (such as COUNT and LEADERELECT) that are functions of the n inputs/ids, by having each node collect all the n inputs/ids [2, 4–7]. Collecting all the n inputs/ids is also explicitly studied as the *token dissemination* problem. (Some of these works [2, 4–7] actually focus on token dissemination, while solving problems such as COUNT as byproduct.) Kuhn et al. [7] have further explored solving COUNT without collecting all n inputs/ids. They propose an elegant randomized algorithm for computing a constant factor approximation for n in $O(n \log \log n)$ time. Different from our algorithm, their algorithm works even for $T = 1$. However, their algorithm's complexity is always $\Omega(n)$ even when $d = O(1)$. The reason is that even when d is small, although the approximation quickly becomes good, the algorithm does not know this, and has to wait for sufficient long before it can make sure. In comparison, our COUNT algorithm

[6] While [10] does not explicitly mention the ∞-interval model, its proofs apply without any change.

has $O(d^3 \log^2 n)$ complexity, is deterministic, and outputs the exact n. On the other hand, our algorithm does not learn all the n ids, and does not solve token dissemination.

Researchers have also considered these distributed computing problems under other settings. Kuhn et al. [19] have studied CONSENSUS, and its variants *coordinated/simultaneous consensus*, in the T-interval model without limits on message sizes. Under such a setting, results from [19] imply that all the problems considered in this paper can be solved deterministically in $O(d)$ rounds. COUNT has been studied in *anonymous* dynamic networks (e.g., [11–16]), but all the algorithms there have $\Omega(n)$ complexity. This is not surprising, since not having unique ids introduces additional challenges that are not present in our context. Among these, the design in [12] also aggregates all values to one node to solve COUNT. Our algorithm uses a similar high-level approach. One of the major differences, however, is that they use random walks to do so, which relies on mixing time and results in $\tilde{O}(n^5)$ complexity. Our algorithm uses explicit aggregation paths, together with a range of other techniques, which eventually achieves $O(d^3 \log^2 n)$ complexity. Our techniques, however, extensively rely on unique node ids, and does not work in anonymous networks. Hence our results and the results from [11–16] are not directly comparable. Some researchers (e.g., [20–22]) have studied LEADERELECT and CONSENSUS in *directed* dynamic networks, which is quite different from our undirected setting. Augustine et al. [23, 24] have studied LEADERELECT and CONSENSUS in dynamic networks with node churn and where the topology is an *expander*. They rely on efficient random walks in expander graphs, which does not apply to our setting. Finally, there have been a body of works (e.g., [25]) on *eventually-stable networks*. The topology of an eventually-stable network may change from round to round, but such changes eventually stop and the algorithm should output sometime after that [26]. In comparison, our algorithms do not wait for the network to stop changing.

Recently, Chlebus et al. [27] have proposed several interesting sublinear algorithms in networks with edge failures. In their model, during the execution of the algorithm, some edges in the network can become *unreliable*. Let G be the topology containing only those reliable edges. Their model allows G to be disconnected. Under such a model, they obtain various upper/lower bounds on the time complexity for solving consensus. In particular, they propose sublinear-time consensus algorithms, with time complexity of either $O(\lambda^3)$ or $O(\lambda)$. Hence λ (unknown to the algorithm) is the *stretch* of G, which is a generalization of the notion of diameter to disconnected graphs. Compared to T-internal dynamic networks in our work, the network model in Chlebus et al. [27] is more general, in the sense that they allow G to be disconnected. When $T = \infty$ in our model, our backbone diameter d is the same as their stretch λ. But our end results and their end results are not comparable: Our sublinear algorithms use messages of size $O(\log n)$, while their sublinear algorithms need messages of size either $O(n \log n)$ or $O(m \log n)$, with m being the number of edges. For $O(\log n)$ message size, *assuming the algorithm has the knowledge of some* λ' *where* $\lambda' \geq \lambda$, they have also designed a consensus algorithm with $O(\lambda')$ complexity. Our algorithms do not need any such knowledge. (Such knowledge has significant impact on the problem: In our model, if d were *known*, then consensus could be trivially solved in $O(d)$ rounds with $O(\log n)$ message size, via simple flooding.) On the other hand, our algorithms do not work, under their more general settings where G can be disconnected. Our techniques are also quite different from theirs.

Finally, a preliminary conference version of this work was previously published as [1], which does not contain any of the proofs in this journal paper. As an update, *after* [1] *had already been published*, and *while this journal paper was under review*, the preliminary conference version [28] of another (but related) research work was published. That later work [28] also aims to obtain sublinear algorithms, for the same problems and same dynamic network model as in this paper. But [28] and this paper are completely different works: They do not build on each other, their algorithms are unrelated and use entirely different techniques, and their results are incomparable. Specifically, at the fundamental level, the algorithms in [28] rely on doing random walks in dynamic networks, while the algorithms in this paper rely on splitting a value across many pre-computed paths between two nodes. Because of this inherent difference, the end results in [28] are incomparable with those in this paper:

- The algorithms in [28] have $O(Dn^{\frac{6}{7}}\text{polylog}(n))$ time complexity. (Here D is the *dynamic diameter* of the network, which is slightly different from the backbone diameter d in this paper.) In contrast, the algorithms in this paper have $O(d^3\text{polylog}(n))$ time complexity. All these algorithms target scenarios where the diameter, D or d, is small. Hence the $O(Dn^{\frac{6}{7}}\text{polylog}(n))$ complexity in [28] is less appealing than the $O(d^3\text{polylog}(n))$ complexity in this paper, since the former still contains a large polynomial term ($n^{\frac{6}{7}}$) of the network size.
- The algorithms in [28] are randomized algorithms with (small) error, while the algorithms in this paper are deterministic and with no error.
- On the other hand, the algorithms in [28] work as long as $T \geq 4$, while the algorithms in this paper need to assume $T \geq c \cdot d^2 \log^2 n$.

2 Overview of Our Approach/Framework and Our Count Algorithm

This section provides the key intuitions behind our approach/framework, while using our Count algorithm as an example. Next, Sect. 3 gives the details of our Count algorithm, and Sect. 4 presents our algorithms for Max/Median/Sum/LeaderElect/Consensus/ConfirmedFlood.

2.1 Starting Point

Let α be the largest id, among all the n nodes in the network. Assume for now that all nodes know α, and we remove this assumption later. In static networks, aggregation is known to be an efficient way to compute Count: We first build a spanning tree rooted at node α. Next each node contributes a value of 1. These values are propagated upstream along the tree, while being aggregated (i.e., summed together) at intermediate tree nodes. Finally, node α gets the sum of all the values, and floods this Count result to all nodes.

In T-interval dynamic networks, however, the changing topology may disrupt the spanning tree. Namely, a tree edge may no longer exist when we need to use it, causing the value from some node u (there can be many such u's) to get stuck somewhere in the middle of the tree path. Imagine that the entire tree aggregation process, including the building of the spanning tree, takes no more than T rounds. By the T-interval model, there must exist some connected spanning subgraph that remains stable in those T rounds. Recall that H_r is the maximum such subgraph for round r through $r + T - 1$. Since H_r is connected, there must exist some path from node u to node α that persists throughout those T rounds. But the problem is that the spanning tree may not contain that particular path—hence the value from u may get stuck in the tree. (If we could magically ensure that the spanning tree only uses edges in H_r, then all problems are solved.) Naively retrying with a different spanning tree does not lead to good complexity.

From the above simple scenario, it is also easy to see that while deleted edges cause problems, newly added edges (i.e., edges that did not exist before but are later created) do not immediately cause any harm. In fact, given that all edges in H_r persist throughout all T rounds, in those T rounds the algorithm can temporarily ignore all edges that are newly added: Even after ignoring those newly added edges, there must still exist a path from every node u to node α. Hence the main challenge will be how to deal with deleted edges.

2.2 Parallel Propagation Over All Paths

Assume for now that the backbone diameter d is known, and we remove this assumption later. (Recall that the COUNT problem is still non-trivial even with known d.) Consider the set L of all paths[7] from some node u to node α with length at most d, in the topology of some round r. Let l be the size of L, which can be exponentially large. Ignore for now the challenge of keeping track of all these paths. Our first key idea is to avoid committing to any specific path for propagating the value. Instead, node u splits its value (of 1.0) into l equal pieces, and propagates each piece along a different path, all in parallel and taking $O(d)$ rounds. We will imagine that a piece "moves" from one node to another along the path, which gives the standard *mass conservation* property [12, 30–32]—the sum of all these pieces on all nodes always remain fixed.

As the pieces are moving, it is possible for most of these l paths to get cut by the adversary—this will cause most pieces to get stuck. But observe that the adversary can stall a piece only if a path existed when we computed the available paths, and then no longer exists when we actually use the path. We can thus effectively limit the adversary's damage by staggering the propagation and by spreading our stake. Specifically, node u propagates its value of 1.0 over x sequential *intervals* (we will set x later), where each interval comprises $O(d)$ rounds and only deals with a value of $\frac{1}{x}$. In each interval, we first *re-compute* the available paths, so that paths that have already been cut before this interval will no longer be considered. We then use those still-available paths to propagate the value. This means that within each interval, node u further breaks the $\frac{1}{x}$ value into many pieces, with each piece corresponding to a distinct path that is still available/alive at the beginning of that interval. (Namely, if

[7] Throughout this paper, we only need to be concerned with paths in a given (static) graph. In dynamic networks, sometimes researchers consider *dynamic paths* [29]—we do not need those.

all l paths are always alive in all x intervals, then the initial value of 1.0 is split into total xl pieces, with l pieces for each interval.) A path cut by the adversary can then only affect a single interval.

Over the course of these x intervals, the number of paths in L that survive may gradually decrease, due to some edges being deleted. Assume that T is no smaller than the total number of rounds in all these x intervals. Then the number of surviving paths can decrease from at most $O(n^d)$ to at least 1. It is "at least 1" because by the definition of the backbone diameter d, the diameter of H_r is at most d and hence H_r must contain a path from node u to node α with length at most d. This path must have survived. Let f_i be the fraction of paths that are alive (i.e., survive) at the end of interval i, among all the paths that were alive at the beginning of interval i. We then have $\prod_{i=1}^{x} f_i \geq \frac{1}{n^d}$. The sum of all those pieces that get stuck is at most $\sum_{i=1}^{x}(1-f_i)\cdot\frac{1}{x}$. By the relation between geometric mean and arithmetic mean, we have $\sum_{i=1}^{x}(1-f_i)\cdot\frac{1}{x} = 1-\frac{1}{x}\sum_{i=1}^{x} f_i \leq 1-(\frac{1}{n^d})^{\frac{1}{x}}$, and the optimal strategy of the adversary to maximize the sum of all those pieces that get stuck is simply to have $f_1 = f_2 = \cdots = f_x$. With $x = d\log n$, the sum $\sum_{i=1}^{x}(1-f_i)\cdot\frac{1}{x}$ will be at most $\frac{1}{2}$. This means that at least half of the value from each node u will successfully reach node α.

At the end of the x intervals, the stuck pieces are simply left in various arbitrary locations across the network. To collect all these stuck pieces, we repeat the above entire process for $2\log n$ times (called $2\log n$ *phases*), where each phase has $x = d\log n$ intervals. (There is no need for T to be larger than the total number of rounds in all phases—T only needs to be larger than the number of rounds in a single phase.) In every phase, each node holding any of the stuck pieces will be viewed as node u in the above discussion. Each such node will start with a value corresponding to the stuck pieces it is holding (instead of starting with a value of 1.0). With such a design, in each phase, node α collects at least half of the remaining value in the network. Since the total count is only n, after the $2\log n$ phases, the leftover will be $n\cdot\frac{1}{n^2}=\frac{1}{n}<1$. Having node α round up the total collected value then gives the exact count n. Finally, note that the algorithm does not know n beforehand, and hence cannot readily compute the two quantities of $d\log n$ and $2\log n$—we deal with this later.

At this point, the above ideas can already enable us to solve COUNT in $O(d^2\log^2 n)$ time complexity: We need total $2\log n$ phases, with each phase having $d\log n$ intervals and each interval having $O(d)$ rounds. The caveat is that we have made a number of significant assumptions along the way. Sections 2.3 and 2.4 will explain how we remove all these assumptions, which entails a collection of non-trivial techniques. One of the techniques will add additional complexity to the algorithm, and our final COUNT algorithm will have $O(d^3\log^2 n)$ time complexity.

Before proceeding, we quickly stress that *a proper balance between intervals and phases* is needed for the idea to work. Without multiple phases, $\Theta(n^2 d\log n)$ intervals would be needed for the leftover to decrease to $\frac{1}{n}$. Without multiple intervals in each phase, $\Theta(n^d\log n)$ phases would be needed. Consider any node u holding some pieces at the beginning of a phase. The key difference between intervals and phases is that for all intervals within that phase, these pieces all start their propagation from the same node u. This allows us to analyze based on the number of surviving paths from node u to node α. While in the next phase, these pieces may start (i.e., continue) their propagation from arbitrary locations in the network.

2.3 Avoiding Excessive Bookkeeping

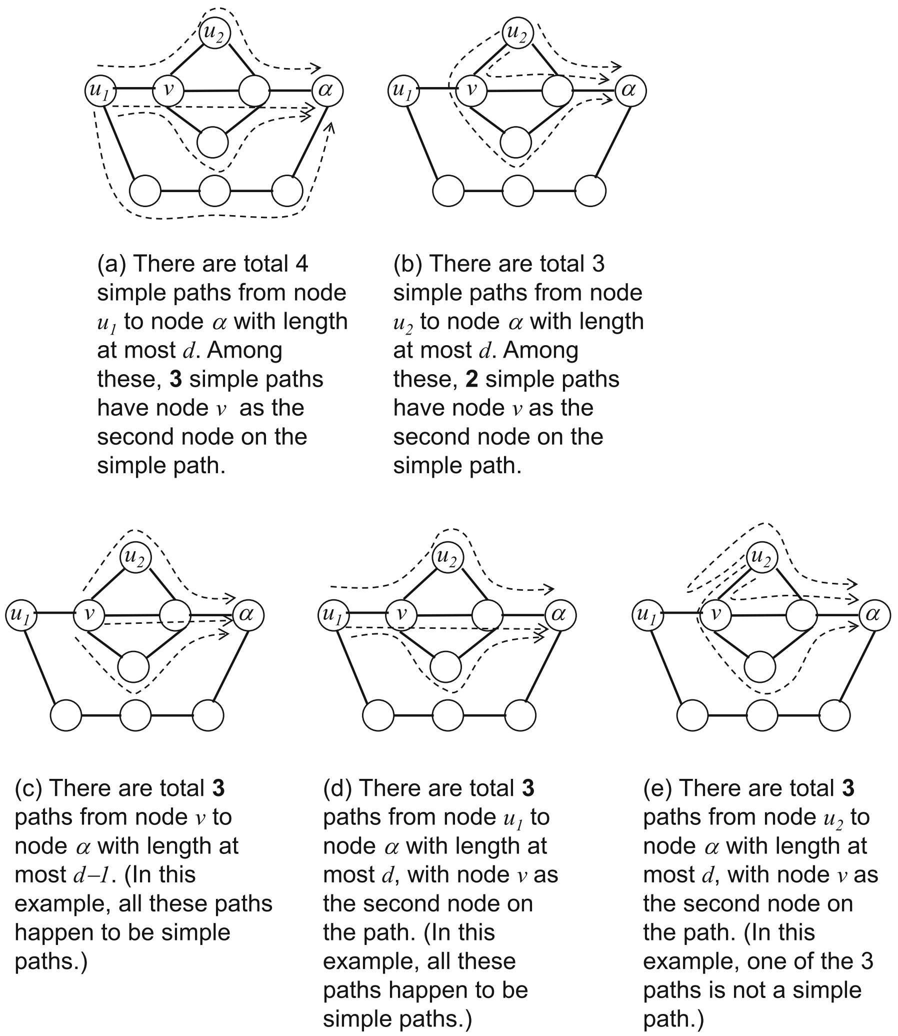
We next overcome the challenge of keeping track of all the pieces and paths. Note that the set L of paths from a given node u to the special node α will change over time. But as explained in Sect. 2.1, if we focus on edge deletions, then L can only shrink over time. When we propagate pieces based on the L computed in a given round r, the shrinkage of L in future rounds will cause some of the pieces to get stuck during propagation. Section 2.2 has already explained how to deal with such stuck pieces. Hence in this section, we only need to focus on the set L in a given round r.

Consider the simple example in Fig. 1, where the network has a *fixed* topology, with $n = 9$ and $d = 4$. Here as shown in Fig. 1a, node u_1 has exactly 4 paths to node α with length at most d, out of which 3 paths have node v as the second node on the path. Imagine that we split the value of 1.0 on node u_1 into 4 equal pieces, and then propagate one piece along each path. Obviously, instead of sending 3 individual pieces to node v where each piece corresponds to $\frac{1}{4}$, node u_1 only needs to send to node v a combined value of $\frac{3}{4}$. Similarly, as shown in Fig. 1b, node u_2 has 3 paths to node α with length at most d, out of which 2 paths have node v as the second node on the path. Hence node u_2 only needs to send to node v a combined value of $\frac{2}{3}$ (corresponding to 2 pieces, each worth $\frac{1}{3}$). But now it is unclear what node v should do: It gets 3 pieces from node u_1 and 2 pieces from node u_2, where different pieces correspond to different values. Each piece needs to follow its own respective path. Obviously, with exponential number of paths and pieces, this becomes tricky.

2.3.1 Allowing Non-distinct Nodes

Interestingly, allowing paths to contain non-distinct nodes (vertices) helps to overcome the above problem. From now on, we use *simple path* (e.g., in Fig. 1a through b) to refer to any path that contains only distinct nodes. We use *path* (e.g., in Fig. 1c through e) in general to refer to any path that may contain multiple occurrences of some nodes. To see why it helps to consider paths instead of simple paths, let us continue with the example in Fig. 1. As shown in Fig. 1c, there are total 3 paths from node v to node α with length at most $d - 1$. Then it must hold, as illustrated in Fig. 1d, that there are exactly 3 paths of length at most d going from node u_1 to node α with v being the second node on the path. By the same reason, there also must be exactly 3 such paths from node u_2 to node α (see Fig. 1e).[8] This means that node v always gets exactly 3 pieces from each of its neighbors. For each set of 3 pieces, node v should forward one piece along each of the 3 paths in Fig. 1c, regardless of which neighbor this set came from. Effectively, the pieces can now be processed in a *memoryless* way. In fact, it suffices for node v to just add up (aggregate) all values it receives from all it neighbors, and send $\frac{1}{3}$ of the total value along each of the 3 paths in Fig. 1c.

[8] In comparison, recall that previously in Fig. 1a, b, the corresponding numbers for *simple paths* were 3 and 2 for node u_1 and u_2, respectively.



(a) There are total 4 simple paths from node u_1 to node α with length at most d. Among these, **3** simple paths have node v as the second node on the simple path.

(b) There are total 3 simple paths from node u_2 to node α with length at most d. Among these, **2** simple paths have node v as the second node on the simple path.

(c) There are total **3** paths from node v to node α with length at most d–1. (In this example, all these paths happen to be simple paths.)

(d) There are total **3** paths from node u_1 to node α with length at most d, with node v as the second node on the path. (In this example, all these paths happen to be simple paths.)

(e) There are total **3** paths from node u_2 to node α with length at most d, with node v as the second node on the path. (In this example, one of the 3 paths is not a simple path.)

Fig. 1 How allowing non-distinct nodes on paths helps to avoid excessive bookkeeping—this figure should be read together with the running example in Sect. 2.3

A further optimization would enable node v to do this by just sending a single message instead of 3 messages, and without node v needing to know whether some of its previous neighbors no longer exist (i.e., due to edge removals). To do so, node v simply sends the total value and the number 3. A neighbor w of node v simply takes a part of the value that is proportional to the number of paths going from node w to node α with length at most $d - 2$. At the same time, since node v also receives a message from each of its current neighbors, node v can simultaneously determine how much value has *not* been taken by its neighbors (due to edge removals). Such leftover value will be kept locally, to be fed into the next phase.

With the above ideas, now we only need to keep track of the *number* of paths going from each node to node α. As a convenient consequence of using paths instead of simple paths, we will be able to easily count such paths efficiently via recursion. (Counting simple paths would instead be #P-Complete.)

2.3.2 Rounding

In the above, when a node transfers a value to another node, precisely describing the value may take too many bits. To avoid this, we carefully round the value so that it takes $O(\log n)$ bits to encode. (We do not know n and hence cannot compute $\log n$—we deal with this later.) The discrepancy between the value and its rounded version will still be kept locally, to be fed into the next phase. (Not discarding the discrepancy is important.) We use similar rounding when counting the number of paths, and will show that such rounding does not compromise correctness.

2.4 Dealing with Unknown *d*, *n*, and α

2.4.1 Unknown *d*

So far we have assumed the knowledge of the backbone diameter d—recall that the algorithm should only use those paths with length at most d, and that the number of intervals in each phase needs to be $\Theta(d \log n)$. To remove this assumption, we use a standard doubling trick to guess d. Let $\tilde{d}$ be our current guess for d. The crux is to determine whether $\tilde{d}$ is too small.

To make such determination, we have node α distribute a predetermined number (i.e., $n^{\tilde{d}}$) of votes to all the nodes over $\tilde{d}$ rounds, where in each round each vote-holding node gives some of its votes to each of its neighbors. (We do not know n—we deal with this later.) This value of $n^{\tilde{d}}$ ensures that if $\tilde{d}$ is large enough (e.g., $\tilde{d} \geq d$), then every node will get at least one vote. Roughly speaking, this is because each node has at most n neighbors, and the number of votes can only get "split" for $\tilde{d}$ times. On the other hand, if $\tilde{d}$ is too small, then some node will not get any vote. Those nodes then force all their neighbors to discard their respective votes. Since the topology is always connected, this causes the total number of remaining votes in the network to be less than $n^{\tilde{d}}$. We next invoke our earlier COUNT algorithm, *for a second time*, to count the remaining votes. (While the algorithm was for counting the number of nodes, it can be easily adapted to count votes, by using $\Theta(d \log n)$ instead of $\Theta(\log n)$ phases.) If $\tilde{d}$ is too small, we will find the vote count to be smaller than $n^{\tilde{d}}$.

Note that the approach may appear circular: When invoking the algorithm to count votes, we again need to know d. But it turns out that if $\tilde{d}$ is too small, our algorithm may undercount the votes but can never overcount. Hence if $\tilde{d}$ is too small, we will always find the vote count to be smaller than $n^{\tilde{d}}$.

2.4.2 Unknown n

While our goal is to compute n, our design so far has relied on the knowledge of n for determining: (i) how many intervals/phases to run (Sect. 2.2), (ii) how many bits to use during rounding (Sect. 2.3), and (iii) how many votes to distribute (Sect. 2.4). One could try to potentially use randomization to first get some rough approximation of n—but we aim for a (stronger) deterministic solution. To this end, we exploit the node ids: Instead of viewing the node ids as opaque, we view them as positive integers. Since (i) there are total n nodes, (ii) all ids are unique, and (iii) the length of each id is $O(\log n)$, we know that the largest id α among the n nodes must satisfy $n \le \alpha \le \text{poly}(n)$. This means that $\log n \le \log \alpha = \Theta(\log n)$. We thus could use $\log \alpha$ in place of $\log n$ for determining the parameters in the above three places, and we can show that doing so does not cause problems. A difficulty, however, is that the algorithm does not actually know α. (If the algorithm were directly given a polynomial upper bound on n, there would be no such difficulty.) In fact, a trivial reduction shows that determining α is at least as hard as LEADERELECT, which is exactly one of the problems we aim to solve.

2.4.3 Unknown α

We finally deal with the fact that the algorithm does not actually know α. First, in the background, we let each node keep sending to its neighbors the largest id that it has seen so far (initially its own id). Next, for any given node $\tilde{\alpha}$, let W be the set of nodes where $\tilde{\alpha}$ is the largest id they have seen so far. The nodes in W will then together run an *instance* of our algorithm. Hence at any point of time, there may be multiple concurrent instances running. Each instance has its own root $\tilde{\alpha}$. When a node in an instance with smaller $\tilde{\alpha}$ learns about another instance with larger $\tilde{\alpha}$, the node *switches* to the latter instance. Switching into or out of an instance in the middle of its execution may cause various technical problems, and hence we only allow nodes to switch in/out at certain specific steps of an instance. We will show, via a careful analysis, that such "delayed" switches will not weaken the asymptotic properties. Finally, with the vote distribution/counting mechanism mentioned earlier, we will ensure that only the instance containing node α can output, when that instance has "grown" to include all nodes. We can then show that such output must be correct.

3 Our $O(d^3 \log^2 n)$ COUNT Algorithm

This section presents the details of our COUNT algorithm. Specifically, Sect. 3.1 gives some definitions and discusses some technicalities. Sections 3.2–3.5 elaborate four building blocks in our COUNT Algorithm. Finally, Sect. 3.6 puts everything together and presents the final COUNT algorithm.

Table 1 Key notations

n	Total number of nodes
d	Backbone diameter
T	Parameter T in T-interval dynamic network model
α	Largest id among the n nodes in the network
$\tilde{d}, \tilde{\alpha}$	Current guesses on d and α, respectively
V	The set of all n nodes in the network
$\sigma(r_1, r_2)$	The maximum spanning subgraph contained by all topologies from round r_1 to r_2 (both inclusive)
$\Gamma_G(u)$	Eccentricity of node u in graph G
$\Lambda_G(u)$	The set of node u's neighbors in graph G
$\Psi_G(m, u, v)$	# of distinct paths from node u to node v in graph G of length exactly m

3.1 Definitions and Technicalities

3.1.1 Definitions

Table 1 summarizes our key notations. As a reminder, at the beginning of Sect. 1, we already defined *T-interval dynamic network*, *backbone diameter*, and *time complexity*. The following gives some more definitions.

Given a T-interval dynamic network, define $\sigma(r_1, r_2)$ to be the maximum (in terms of number of edges) spanning subgraph contained by all the topologies in all rounds from r_1 to r_2 (both inclusive). Given a node id x, we use "x" to refer to the integer value of x, and "node x" to refer to that node. Without loss of generality, we assume that node ids are never smaller than 2. We use V to denote the set of all nodes in the network, where $|V| = n$. We use α to denote the largest node id among the n nodes in the network. Recall that d is the backbone diameter of the dynamic network. We use $\tilde{d}$ and $\tilde{\alpha}$ to denote a node's current guesses on d and α, respectively. Without loss of generality, in each round, we add a self-loop to each node in the topology—hence every node is also a neighbor of itself, and receives its own message. For a given graph G (with self-loops), a *path* of length m is a sequence of nodes $x_0, x_1, \ldots, x_m$ such that x_{i-1} is a neighbor of x_i for all $i \in [1, m]$. In particular, a path may contain multiple occurrences of the same node. With self-loops, we will only need to consider paths of length exactly d, rather than at most d. Let $\Gamma_G(u)$ be the eccentricity of node u in G, and $\Gamma_G(u) = \infty$ if G is disconnected. Given graph G and node u, let $\Lambda_G(u)$ be the set of neighbors of node u in G. Given graph G, integer $m \geq 1$, and any two nodes u and v, define $\Psi_G(m, u, v)$ to be the number of distinct paths from node u to node v in G of length exactly m. Define $\Psi_G(0, u, v) = 1$ if $u = v$, and $\Psi_G(0, u, v) = 0$ if $u \neq v$.

Recall from Sect. 2.4 that there may be multiple concurrent instances running in the network. We say that *node v does not interfere with instance $\tilde{\alpha}$ in round r* iff node v in round r only sends messages of the form $\langle x, y, \ldots \rangle$ where either $x \in \{\texttt{SWITCH}, \texttt{OUTPUT}\}$ or $y \neq \tilde{\alpha}$.

3.1.2 Newcomer Messages and Oldcomer Messages

In our algorithm, each node always sends a message in each round. Consider any given node u. Throughout our COUNT algorithm (and across all the invocations of the various subroutines), node u maintains a global variable S, which is initialized to the set of all positive integers. (Obviously, one can use finite data structures to achieve what we need for S here.) At the end of each round, node u receives a set of messages. A message whose sender's id is not in S is a *newcomer message*, otherwise it is an *oldcomer message*. Unless otherwise mentioned in the algorithm, node u will always ignore newcomer messages. After processing all the messages in this round, node u immediately updates S to be the ids of all the senders of the oldcomer messages received in this round. Finally, at the very beginning of each round, node u has the option of resetting S to the set of all positive integers, by invoking `ResetNeighbors()`.

Intuitively, the above mechanism enables node u to temporarily ignore messages coming from newly created (or newly recovered) edges in the network, until the next time that node u invokes `ResetNeighbors()`. For clarity, our pseudo-codes do not explicit mention that newcomer messages are ignored. (We will, of course, clearly indicate when to invoke `ResetNeighbors()`.) In several special places, our algorithm does not ignore newcomer messages, which we will clearly indicate.

3.1.3 Rounding

Section 2.3 mentioned that our algorithm sometimes (e.g., when a node transfers a value to another node) uses rounding to avoid needing more than $O(\log n)$ bits in each message. The following explains how exactly such rounding is done. Consider any real value $x \geq 0$, and consider the id $\tilde{\alpha}$ of any node in the network. (We must have $\tilde{\alpha} \leq \alpha \leq \text{poly}(n)$.) When $x > 0$, let the unique real value a and integer b be such that $1 \leq a < 2$ and $x = a2^b$. Our algorithm later will need to use both a "rounded-up" version of x and a "rounded-down" version of x.

For the "rounded-up" version, our algorithm will be concerned with $x \in [1, 2^{(n\tilde{\alpha})^4}]$. (The algorithm actually may also encounter the case of $x = 0$. But obviously, this special case can be separately encoded using one extra bit.) For $x \in [1, 2^{(n\tilde{\alpha})^4}]$, the corresponding b value will be in $[0, (n\tilde{\alpha})^4]$, and can already be encoded using $O(\log n)$ bits. So we only need to properly round the a value. To do so, we discretize a using a granularity of $\frac{1}{\tilde{\alpha}^6}$. (We do not use a granularity of $1/\text{poly}(n)$ because the algorithm does not know n. In comparison, $\tilde{\alpha}$ will be explicitly maintained and hence known by the algorithm.) Specifically, let a^+ be the smallest value such that $a^+ \geq a$ and a^+ is a multiple of $\frac{1}{\tilde{\alpha}^6}$. We define the "rounded-up" version of x as $\texttt{round}^+(x, \tilde{\alpha}) = a^+ \cdot 2^b$. For convenience, we also define $\texttt{round}^+(0, \tilde{\alpha}) = 0$. Obviously, we have $x \leq \texttt{round}^+(x, \tilde{\alpha}) \leq \left(1 + \frac{1}{\tilde{\alpha}^6}\right) x$. For all $x \in [1, 2^{(n\tilde{\alpha})^4}]$, the value of $\texttt{round}^+(x, \tilde{\alpha})$ can be encoded using $O(\log n)$ bits: To encode $\texttt{round}^+(x, \tilde{\alpha}) = a^+ \cdot 2^b$, we only need to specify three integers b, $\tilde{\alpha}^6$, and $a^+ \cdot \tilde{\alpha}^6$, taking only $O(\log \log 2^{(n\tilde{\alpha})^4} + \log \tilde{\alpha}) = O(\log n)$ bits.

For the "rounded-down" version, our algorithm will be concerned with $x \in [0, 2^{(n\tilde{\alpha})^4}]$. We again use a granularity of $\frac{1}{\tilde{\alpha}^6}$. To avoid dealing with a too small value for b when x is close to 0, we define the "rounded-down" version of x as $\texttt{round}^{-}(x, \tilde{\alpha}) = 0$ for $x \in \left[0, \frac{1}{\tilde{\alpha}^6}\right)$. For $x \in \left[\frac{1}{\tilde{\alpha}^6}, 2^{(n\tilde{\alpha})^4}\right]$, we let a^- be the largest value such that $a^- \leq a$ and a^- is a multiple of $\frac{1}{\tilde{\alpha}^6}$, and we define the "rounded-down" version of x as $\texttt{round}^{-}(x, \tilde{\alpha}) = a^- \cdot 2^b$. For all $x \in [0, 2^{(n\tilde{\alpha})^4}]$, one can easily verify that $(1 - \frac{1}{\tilde{\alpha}^6})x - \frac{1}{\tilde{\alpha}^6} \leq \texttt{round}^{-}(x, \tilde{\alpha}) \leq x$, and that the value of $\texttt{round}^{-}(x, \tilde{\alpha})$ can be encoded using $O(\log n)$ bits.

3.2 Building Block: Counting Paths

Algorithm 1 gives the subroutine for counting the number of distinct paths via a simple recursion. The time complexity of the algorithm will be much smaller than T, under our invocation parameters later and when $T \geq cd^2 \log^2 n$. Line 5 of Algorithm 1 sets $l_i \leftarrow \texttt{round}^{+}(x, \tilde{\alpha})$. Here it is important to round the value up, instead of down. This means that if a node's neighbors collectively have x paths to the root, then the node itself will claim that it has x' paths to the root, where $x' \geq x$. In turn, during value propagation later, conceptually the node will ask for total x' pieces, so that it ensures that it has x pieces available to propagate to its neighbors. Those $x' - x$ unpropagated pieces will simply be kept locally on the node. The network topology may change during the execution of Algorithm 1, potentially decreasing the number of paths as we go. This will not cause any problem, and no special mechanism is needed to avoid this. (Algorithm 1 never invokes `ResetNeighbors`(), hence the number of paths seen by the algorithm does not increase.) Recall that V denotes the set of all nodes in the network. Lemma 1 proves the guarantees of `CountPaths()` (i.e., Algorithm 1):

Lemma 1 (Guarantees of Algorithm 1) *Consider any rounds r' and r where $r' \leq r$, any node $\tilde{\alpha}$, and any integer $\tilde{d}$ where $2 \leq \tilde{d} \leq \tilde{\alpha}$. Let W be any set of nodes that (i) invoke* `CountPaths`$(\tilde{\alpha}, \tilde{d})$ *(i.e., Algorithm 1) simultaneously in round r, and (ii) last invoked* `ResetNeighbors`() *at the beginning of round r'. Let $G_1 = \sigma(r', r)$ and $G_2 = \sigma(r', r + \tilde{d})$. Let $l_{i,u}$ be the l_i value output by Algorithm 1 on node u, for $i \in [0, \tilde{d}]$ and $u \in W$. If no node in $V \backslash W$ interferes with instance $\tilde{\alpha}$ in any round from round r to round $r + \tilde{d} - 1$ (both inclusive), then all the following holds for all $i \in [0, \tilde{d}]$ and all $u \in W$:*

$$l_{i,u} \geq \sum_{v \in (\Lambda_{G_2}(u) \cap W)} l_{i-1,v} \qquad \textit{for} \quad i \neq 0 \tag{1}$$

$$0 \leq l_{i,u} \leq \left(1 + \frac{1}{\tilde{\alpha}^4}\right) \Psi_{G_1}(i, u, \tilde{\alpha}) \tag{2}$$

$$l_{i,u} \geq \Psi_{G_2}(i, u, \tilde{\alpha}) \qquad \textit{if} \quad W = V \tag{3}$$

Finally, for all i and u, the value $l_{i,u}$ can be encoded using $O(\log n)$ bits, and node u always sends $O(\log n)$ bits in every round of its execution of Algorithm 1.

Algorithm 1 CountPaths.
Input: $\tilde{\alpha}$ (guess on α) and $\tilde{d}$ (guess on d);
Output: Each node u outputs, for $i \in [0, \tilde{d}]$, the number of paths of length exactly i from node u to node $\tilde{\alpha}$.

```
1: procedure CountPaths($\tilde{\alpha}, \tilde{d}$)
2:     if $\tilde{\alpha}$ = my id then $l_0 \leftarrow 1$; else $l_0 \leftarrow 0$;
3:     for $i \leftarrow 1, \ldots, \tilde{d}$ do                ▷ This loop takes total $\tilde{d}$ rounds.
4:         send ⟨COUNT_PATH, $\tilde{\alpha}, l_{i-1}$⟩;
5:         $l_i \leftarrow \texttt{round}^+(x, \tilde{\alpha})$, where $x$ is the sum of all the $l'_{i-1}$ values in all the ⟨COUNT_PATH, $\tilde{\alpha}', l'_{i-1}$⟩
   messages I receive this round with $\tilde{\alpha}' = \tilde{\alpha}$;
6:     return $l_0, l_1, l_2, \ldots, l_{\tilde{d}}$;
```

Proof To prove Eq. (1), let $G_3 = \sigma(r', r+i-1)$. Since $\Lambda_{G_2}(u) \subseteq \Lambda_{G_3}(u)$, we have:

$$\begin{aligned} l_{i,u} &= \texttt{round}^+\left(\sum_{v \in (\Lambda_{G_3}(u) \cap W)} l_{i-1,v}, \tilde{\alpha}\right) \\ &\geq \texttt{round}^+\left(\sum_{v \in (\Lambda_{G_2}(u) \cap W)} l_{i-1,v}, \tilde{\alpha}\right) \\ &\geq \sum_{v \in (\Lambda_{G_2}(u) \cap W)} l_{i-1,v} \end{aligned}$$

Equation (1), together with the fact that $l_{0,v} \geq 0$ for all $v \in W$, implies that $l_{i,u} \geq 0$. For Eq. (2), now we only need to prove $l_{i,u} \leq \left(1 + \frac{1}{\tilde{\alpha}^4}\right) \Psi_{G_1}(i, u, \tilde{\alpha})$. We will show via an induction that $l_{i,u} \leq \left(1 + \frac{1}{\tilde{\alpha}^6}\right)^i \Psi_{G_1}(i, u, \tilde{\alpha})$, which implies that $l_{i,u} \leq \left(1 + \frac{1}{\tilde{\alpha}^6}\right)^{\tilde{\alpha}} \Psi_{G_1}(i, u, \tilde{\alpha}) \leq \left(1 + \frac{1}{\tilde{\alpha}^4}\right) \Psi_{G_1}(i, u, \tilde{\alpha})$. The induction base for $i = 0$ is trivial. Suppose the claim holds for $i = j$. Let $G_4 = \sigma(r', r+j)$. Since $\Lambda_{G_4}(u) \subseteq \Lambda_{G_1}(u)$, by the induction hypothesis and by the property of $\texttt{round}^+()$, we have:

$$\begin{aligned} l_{j+1,u} &= \texttt{round}^+\left(\sum_{v \in (\Lambda_{G_4}(u) \cap W)} l_{j,v}, \tilde{\alpha}\right) \leq \texttt{round}^+\left(\sum_{v \in (\Lambda_{G_1}(u) \cap W)} l_{j,v}, \tilde{\alpha}\right) \\ &\leq \left(1 + \frac{1}{\tilde{\alpha}^6}\right) \sum_{v \in (\Lambda_{G_1}(u) \cap W)} l_{j,v} \leq \left(1 + \frac{1}{\tilde{\alpha}^6}\right)^{j+1} \sum_{v \in (\Lambda_{G_1}(u) \cap W)} \Psi_{G_1}(j, v, \tilde{\alpha}) \\ &\leq \left(1 + \frac{1}{\tilde{\alpha}^6}\right)^{j+1} \sum_{v \in \Lambda_{G_1}(u)} \Psi_{G_1}(j, v, \tilde{\alpha}) = \left(1 + \frac{1}{\tilde{\alpha}^6}\right)^{j+1} \Psi_{G_1}(j+1, u, \tilde{\alpha}) \end{aligned}$$

Next, we use an induction on i to directly prove Eq. (3). The induction base for $i = 0$ is trivial. Suppose the claim holds for $i = j$. Let $G_4 = \sigma(r', r +$

j). Since $\Lambda_{G_2}(u) \subseteq \Lambda_{G_4}(u)$, together with the induction hypothesis, we have $l_{j+1,u} = \texttt{round}^{+}(\sum_{v\in(\Lambda_{G_4}(u)\cap W)} l_{j,v}, \tilde{\alpha}) = \texttt{round}^{+}(\sum_{v\in\Lambda_{G_4}(u)} l_{j,v}, \tilde{\alpha}) \geq \texttt{round}^{+}(\sum_{v\in\Lambda_{G_2}(u)} l_{j,v}, \tilde{\alpha}) \geq \sum_{v\in\Lambda_{G_2}(u)} l_{j,v} \geq \sum_{v\in\Lambda_{G_2}(u)} \Psi_{G_2}(j, v, \tilde{\alpha}) = \Psi_{G_2}(j+1, u, \tilde{\alpha})$.

Finally, we show that $l_{i,u}$ can be encoded using $O(\log n)$ bits. Note that $l_{i,u}$ is assigned a value (i..e, $\texttt{round}^{+}(x, \tilde{\alpha})$) only once at Line 5 of Algorithm 1. One can easily verify based on the pseudo-code that at Line 5, either $x = 0$ or $x \geq 1$. If $x = 0$, then $l_{i,u} = \texttt{round}^{+}(x, \tilde{\alpha}) = 0$ can be encoded using a single bit. If $x \geq 1$, Eq. (2) tells us that $x \leq l_{i,u} \leq 2\Psi_{G_1}(i, u, \tilde{\alpha}) \leq 2n^{i} \leq 2n^{\tilde{d}} \leq 2n^{\tilde{\alpha}} \leq 2^{(n\tilde{\alpha})^4}$, implying that $l_{i,u} = \texttt{round}^{+}(x, \tilde{\alpha})$ can be encoded using $O(\log n)$ bits. Therefore $l_{i,u}$ can always be encoded using $1 + \max(1, O(\log n)) = O(\log n)$ bits. A node $u \in W$, during its execution of Algorithm 1, only send a message at Line 4. Each such message contains a label `COUNT_PATH`, a node id $\tilde{\alpha}$, and $l_{i,u}$, all of which can be encoded using $O(\log n)$ bits. □

3.3 Building Block: Aggregating within an Interval

Algorithm 2 gives the subroutine for aggregating the values, within an interval. Each interval here consists of $2\tilde{d}$ rounds. Under invocation parameters later and when $T \geq cd^2 \log^2 n$, the value $2\tilde{d}$ will be much smaller than T. Recall from Sect. 2 that each node splits its value `itv_input` across all the paths, and propagates all those pieces to node $\tilde{\alpha}$ in parallel, with proper aggregation along the way. Algorithm 2 first invokes Algorithm 1 to count paths. For any node u, let $l_{i,u}$ be the computed number of paths from node u to node $\tilde{\alpha}$ with length exactly i. Next the algorithm goes through $\tilde{d}$ steps. Intuitively, each step moves all pieces simultaneously one step further along their respective paths. Consider any node u and any neighboring node v of node u. In the very first step, node u sends a message containing the rounded-down version of its initial value `itv_input`. (We round down so that a node never sends more than what it has.) Doing so transfers a $\frac{l_{\tilde{d}-1,v}}{l_{\tilde{d},u}}$ fraction[9] of `itv_input` from node u to node v (and also transfers some other fractions from node u to its other neighbors). This quantity should then be added to the value on node v and subtracted from the value on node u. The remaining steps are similar: In the i-th step ($i \geq 2$), each node sends its current local value, and we use the fraction $\frac{l_{\tilde{d}-i,v}}{l_{\tilde{d}-i+1,u}}$ instead of $\frac{l_{\tilde{d}-1,v}}{l_{\tilde{d},u}}$.

Lemma 2 next proves the guarantees of `IntervalAggregate()` (i.e., Algorithm 2). In the lemma, Eq. (5) is the mass conservation property, while Eq. (6) shows that the fraction of total mass moved to $\tilde{\alpha}$ is proportional to the fraction of surviving paths. We prove the lemma by showing that in the ith step, the value on any node u is proportional to the fraction of surviving paths from each node v to node $\tilde{\alpha}$ with node u being the ith vertex on that path. For the purpose of such reasoning, a "surviving" path would mean that the part from node v to node u is still intact, regardless of whether the

[9] If $l_{\tilde{d},u} = 0$, then node u has no path of length $\tilde{d}$ to $\tilde{\alpha}$. This necessarily implies that node v has no path of length $\tilde{d} - 1$ to $\tilde{\alpha}$, and hence $l_{\tilde{d}-1,v} = 0$ as well. In such a case, node u will not transfer any value to node v. Hence we define $\frac{0}{0} = 0$ here.

Algorithm 2 `IntervalAggregate`.
/* Let z be the sum of all the `itv_input` values on all the invoking nodes. This subroutine aims to collect z to node $\tilde{\alpha}$ as much as possible, while leaving the leftovers on the other nodes. */
Input: $\tilde{\alpha}$ (guess on α), $\tilde{d}$ (guess on d), and `itv_input` (real value);
Output: Node $\tilde{\alpha}$ outputs the total value it has collected. Other nodes output their respective leftover values.

```
1: procedure IntervalAggregate($\tilde{\alpha}, \tilde{d}$, itv_input)
2:   $l_0, l_1, \ldots, l_{\tilde{d}} \leftarrow$ CountPaths($\tilde{\alpha}, \tilde{d}$);        ▷ CountPaths() takes total $\tilde{d}$ rounds.
3:   remain ← itv_input;
4:   for $i \leftarrow 1 \ldots \tilde{d}$ do                                  ▷ This loop takes total $\tilde{d}$ rounds.
5:     rounded ← round⁻(remain, $\tilde{\alpha}$);
6:     send ⟨AGGREGATE, $\tilde{\alpha}$, rounded, $l_{\tilde{d}-i+1}, l_{\tilde{d}-i}$⟩;
7:     for every ⟨AGGREGATE, $\tilde{\alpha}'$, rounded′, $l'_{\tilde{d}-i+1}, l'_{\tilde{d}-i}$⟩ message received in this round with $\tilde{\alpha}' = \tilde{\alpha}$ do
8:       remain ← remain + $\frac{l_{\tilde{d}-i}}{l'_{\tilde{d}-i+1}}$ · rounded′ − $\frac{l'_{\tilde{d}-i}}{l_{\tilde{d}-i+1}}$ · rounded;
9:       // In the above, $\frac{0}{0}$ is defined to be 0.
10:    end for
11:  end for
12:  return remain;
```

part from node u to node $\tilde{\alpha}$ is still intact. This careful reasoning is needed — otherwise we would be overly pessimistic and the proof would not go through. Again, recall that V denotes the set of all nodes in the network.

Lemma 2 (Guarantees of Algorithm 2) *Consider any rounds r' and r where $r' \leq r$, any node $\tilde{\alpha}$, and any integer $\tilde{d}$ where $2 \leq \tilde{d} \leq \tilde{\alpha}$. Let W be any set of nodes where $\tilde{\alpha} \in W$ and where each node $u \in W$ (i) invokes* `IntervalAggregate`($\tilde{\alpha}, \tilde{d}$, `itv_input`$_u$) *(i.e., Algorithm 2) in round r with some* `itv_input`$_u \geq 0$ *such that* $\sum_{u\in W}$ `itv_input`$_u \leq n + \tilde{\alpha}^{\tilde{d}}$*, and (ii) last invoked* `ResetNeighbors()` *at the beginning of round r'. Let $G_1 = \sigma(r', r)$ and $G_2 = \sigma(r', r + 2\tilde{d})$. Let* `output`$_u$ *be the return value of Algorithm 2 on node u for $u \in W$. If no node in $V \setminus W$ interferes with instance $\tilde{\alpha}$ in any round from round r to round $r + 2\tilde{d} - 1$ (both inclusive), then all the following holds:*

$$\texttt{output}_u \geq 0, \quad \textit{for all} \quad u \in W \tag{4}$$

$$\sum_{u\in W} \texttt{output}_u = \sum_{u\in W} \texttt{itv\_input}_u \tag{5}$$

$$\texttt{output}_{\tilde{\alpha}} \geq \frac{3}{4} \sum_{v\in V} \left(\frac{\Psi_{G_2}(\tilde{d}, v, \tilde{\alpha})}{\Psi_{G_1}(\tilde{d}, v, \tilde{\alpha})} \cdot \texttt{itv\_input}_v \right) - \frac{n}{\tilde{\alpha}^5}, \quad \textit{if } W = V \textit{ and } \tilde{d} \geq \Gamma_{G_2}(\tilde{\alpha}) \tag{6}$$

Finally, for any $u \in W$, node u always sends $O(\log n)$ bits in every round of its execution of Algorithm 2.

Proof All line numbers in this proof refer to Algorithm 2. For any $u \in W$, let $l_{0,u}, l_{1,u}, \ldots, l_{\tilde{d},u}$ denote the values of $l_0, l_1, \ldots l_{\tilde{d}}$, respectively, on node u immediately after Line 2. Consider the iteration from Line 5 to 10 (both inclusive). For $i \in [1, \tilde{d}]$, let $\texttt{remain}_{i,u}$ be the value of $\texttt{remain}$ at the end of iteration i on node u. Let $\texttt{remain}_{0,u} = \texttt{itv\_input}_u$. Recall that Algorithm 2 defines $\frac{0}{0}$ to be 0. We also define $\frac{0}{0}$ to be 0 in this proof. One can (tediously) verify that in both Algorithm 2 and our proof next, we will never encounter a quantity of $\frac{x}{0}$ with $x \neq 0$.

We first use a simple induction to prove $\texttt{remain}_{i,u} \geq 0$ for all $i \in [0, \tilde{d}]$, which implies Eq. (4). The induction base is trivial. Suppose $\texttt{remain}_{i,u} \geq 0$ holds for $i = j$, and we now prove for $j + 1$. Let $G_3 = \sigma(r', r + \tilde{d})$ and $G_4 = \sigma(r', r + \tilde{d} + j)$. By Eq. (1) in Lemma 1, we have $\sum_{w \in (\Lambda_{G_4}(u) \cap W)} \frac{l_{\tilde{d}-j-1,w}}{l_{\tilde{d}-j,u}} \leq \sum_{w \in (\Lambda_{G_3}(u) \cap W)} \frac{l_{\tilde{d}-j-1,w}}{l_{\tilde{d}-j,u}} \leq 1$. Hence:

$$\begin{aligned} \texttt{remain}_{j+1,u} &\geq \texttt{remain}_{j,u} - \left(\sum_{w \in (\Lambda_{G_4}(u) \cap W)} \frac{l_{\tilde{d}-j-1,w}}{l_{\tilde{d}-j,u}} \right) \\ &\quad \cdot \texttt{round}^{-}(\texttt{remain}_{j,u}, \tilde{\alpha}) \\ &\geq \texttt{remain}_{j,u} - \texttt{round}^{-}(\texttt{remain}_{j,u}, \tilde{\alpha}) \geq 0 \end{aligned} \tag{7}$$

We next prove $\sum_{u \in W} \texttt{remain}_{i,u} = \sum_{u \in W} \texttt{itv\_input}_u$ for all $i \in [0, \tilde{d}]$, which implies Eq. (5). Obviously $\sum_{u \in W} \texttt{remain}_{0,u} = \sum_{u \in W} \texttt{itv\_input}_u$. The value of $\sum_{u \in W} \texttt{remain}_{1,u}$ is fully determined in the first iteration, which is in round $r + \tilde{d}$. Consider any given $u \in W$, and any node v that is a neighbor of node u in round $r + \tilde{d}$. If $v \notin W$, then by the condition in the lemma, node v does not interfere with instance $\tilde{\alpha}$ in that round. By the pseudo-code, the message from node v to node u will be ignored by the algorithm, and has no effect on $\texttt{remain}_{1,u}$. If $v \in W$, then the messages exchanged between node u and node v in that round will cause $\texttt{remain}_{1,u}$ to increase by $\frac{l_{\tilde{d}-i,u}}{l_{\tilde{d}-i+1,v}} \cdot \texttt{round}^{-}(\texttt{remain}_{i-1,v}, \tilde{\alpha}) - \frac{l_{\tilde{d}-i,v}}{l_{\tilde{d}-i+1,u}} \cdot \texttt{round}^{-}(\texttt{remain}_{i-1,u}, \tilde{\alpha})$, while causing $\texttt{remain}_{1,v}$ to decrease by the same amount. Hence, $\sum_{u \in W} \texttt{remain}_{1,u}$ will remain unchanged after the message exchange between node u and node v. Putting both cases together, we have $\sum_{u \in W} \texttt{remain}_{1,u} = \sum_{u \in W} \texttt{remain}_{0,u} = \sum_{u \in W} \texttt{itv\_input}_u$. The same argument generalizes to $\sum_{u \in W} \texttt{remain}_{i,u}$ for all $i \in [0, \tilde{d}]$.

We move on to prove Eq. (6), when $W = V$ and $\tilde{d} \geq \Gamma_{G_2}(\tilde{\alpha})$. We will prove that when $W = V$ and $\tilde{d} \geq \Gamma_{G_2}(\tilde{\alpha})$, for all $i \in [0, \tilde{d}]$ and all $u \in V$, there exist non-negative real values $y_{i,u}$ such that:

$$\sum_{v \in V} y_{i,v} \leq \frac{in}{\tilde{\alpha}^6} \tag{8}$$

$$\texttt{remain}_{i,u} \geq \frac{\left(1 - \frac{1}{\tilde{\alpha}^6}\right)^i}{1 + \frac{1}{\tilde{\alpha}^4}} \sum_{v \in V} \left(\frac{\Psi_{G_2}(i, v, u) \cdot l_{\tilde{d}-i,u}}{\Psi_{G_1}(\tilde{d}, v, \tilde{\alpha})} \cdot \texttt{itv\_input}_v \right) - y_{i,u} \tag{9}$$

Equation (6) directly follows from Eqs. (8) and 9 since:

$$\begin{aligned}
\texttt{output}_{\tilde\alpha} &= \texttt{remain}_{\tilde d,\tilde\alpha} \\
&\geq \frac{\left(1-\frac{1}{\tilde\alpha^6}\right)^{\tilde d}}{1+\frac{1}{\tilde\alpha^4}} \cdot \sum_{v\in V}\left(\frac{\Psi_{G_2}(\tilde d, v, \tilde\alpha)\cdot l_{0,\tilde\alpha}}{\Psi_{G_1}(\tilde d, v, \tilde\alpha)}\cdot \texttt{itv\_input}_v\right) - y_{\tilde d,\tilde\alpha} \\
&\geq \frac{\left(1-\frac{1}{\tilde\alpha^6}\right)^{\tilde\alpha}}{1+\frac{1}{\tilde\alpha^4}} \sum_{v\in V}\left(\frac{\Psi_{G_2}(\tilde d, v, \tilde\alpha)}{\Psi_{G_1}(\tilde d, v, \tilde\alpha)}\cdot \texttt{itv\_input}_v\right) - y_{\tilde d,\tilde\alpha} \\
&\geq (\text{since } 2\leq\tilde\alpha)\frac{3}{4}\sum_{v\in V}\left(\frac{\Psi_{G_2}(\tilde d, v, \tilde\alpha)}{\Psi_{G_1}(\tilde d, v, \tilde\alpha)}\cdot \texttt{itv\_input}_v\right) - y_{\tilde d,\tilde\alpha} \\
&\geq \frac{3}{4}\sum_{v\in V}\left(\frac{\Psi_{G_2}(\tilde d, v, \tilde\alpha)}{\Psi_{G_1}(\tilde d, v, \tilde\alpha)}\cdot \texttt{itv\_input}_v\right) - \frac{n}{\tilde\alpha^5}
\end{aligned}$$

We next prove Eqs. (8) and (9) via an induction. For $i=0$, we set $y_{0,v}=0$ for all v. By Lemma 1 we have $\left(\frac{1}{1+\frac{1}{\tilde\alpha^4}}\right)\cdot\sum_{v\in V}\left(\frac{\Psi_{G_2}(0,v,u)\cdot l_{\tilde d,u}}{\Psi_{G_1}(\tilde d,v,\tilde\alpha)}\cdot\texttt{itv\_input}_v\right)-y_{0,u}=\frac{1}{1+\frac{1}{\tilde\alpha^4}}\frac{l_{\tilde d,u}}{\Psi_{G_1}(\tilde d,u,\tilde\alpha)}\cdot\texttt{itv\_input}_u\leq\texttt{itv\_input}_u=\texttt{remain}_{0,u}$. Next assume the claim holds for $i=j$. Let $G_3=\sigma(r',r+\tilde d)$ and $G_4=\sigma(r',r+\tilde d+j)$. We set $y_{j+1,u}=\sum_{w\in\Lambda_{G_4}(u)}\left(\frac{l_{\tilde d-j-1,u}}{l_{\tilde d-j,w}}\left(\frac{1}{\tilde\alpha^6}+y_{j,w}\right)\right)$. For Eq. (8), we have:

$$\begin{aligned}
\sum_{v\in V} y_{j+1,v} &= \sum_{v\in V}\sum_{w\in\Lambda_{G_4}(v)}\left(\frac{l_{\tilde d-j-1,v}}{l_{\tilde d-j,w}}\left(\frac{1}{\tilde\alpha^6}+y_{j,w}\right)\right) \\
&= \sum_{w\in V}\sum_{v\in\Lambda_{G_4}(w)}\left(\frac{l_{\tilde d-j-1,v}}{l_{\tilde d-j,w}}\left(\frac{1}{\tilde\alpha^6}+y_{j,w}\right)\right) \\
&\leq \sum_{w\in V}\sum_{v\in\Lambda_{G_3}(w)}\left(\frac{l_{\tilde d-j-1,v}}{l_{\tilde d-j,w}}\left(\frac{1}{\tilde\alpha^6}+y_{j,w}\right)\right) \\
&\leq \sum_{w\in V}\left(\frac{1}{\tilde\alpha^6}+y_{j,w}\right) \text{ (by Lemma 1 and since } W=V) \\
&\leq \frac{n}{\tilde\alpha^6}+\frac{jn}{\tilde\alpha^6} \text{ (by inductive hypothesis) } = \frac{(j+1)n}{\tilde\alpha^6}
\end{aligned}$$

To prove Eq. (9), note that by Eq. (7) and since $W=V$, we have $0\leq\texttt{remain}_{j,u}-\left(\sum_{w\in\Lambda_{G_4}(u)\cap W}\frac{l_{\tilde d-j-1,w}}{l_{\tilde d-j,u}}\right)\cdot\texttt{round}^{-}(\texttt{remain}_{j,u},\tilde\alpha)=\texttt{remain}_{j,u}-$

$\left(\sum_{w\in\Lambda_{G_4}(u)} \frac{l_{\tilde{d}-j-1,w}}{l_{\tilde{d}-j,u}}\right)\cdot\texttt{round}^{-}(\texttt{remain}_{j,u},\tilde{\alpha})$. Hence (and again because $W = V$), we have:

$$\begin{aligned}
\texttt{remain}_{j+1,u} &= \texttt{remain}_{j,u} - \left(\sum_{w\in\Lambda_{G_4}(u)} \frac{l_{\tilde{d}-j-1,w}}{l_{\tilde{d}-j,u}}\right)\cdot\texttt{round}^{-}(\texttt{remain}_{j,u},\tilde{\alpha}) \\
&\quad + \sum_{w\in\Lambda_{G_4}(u)}\left(\frac{l_{\tilde{d}-j-1,u}}{l_{\tilde{d}-j,w}}\cdot\texttt{round}^{-}(\texttt{remain}_{j,w},\tilde{\alpha})\right) \\
&\geq \sum_{w\in\Lambda_{G_4}(u)}\left(\frac{l_{\tilde{d}-j-1,u}}{l_{\tilde{d}-j,w}}\cdot\texttt{round}^{-}(\texttt{remain}_{j,w},\tilde{\alpha})\right) \\
&\geq \sum_{w\in\Lambda_{G_4}(u)}\left(\frac{l_{\tilde{d}-j-1,u}}{l_{\tilde{d}-j,w}}\left(\left(1-\frac{1}{\tilde{\alpha}^6}\right)\texttt{remain}_{j,w}-\frac{1}{\tilde{\alpha}^6}\right)\right) \\
&\geq \sum_{w\in\Lambda_{G_4}(u)}\left(\frac{l_{\tilde{d}-j-1,u}}{l_{\tilde{d}-j,w}}\left(\left(1-\frac{1}{\tilde{\alpha}^6}\right)\left(\frac{\left(1-\frac{1}{\tilde{\alpha}^6}\right)^j}{1+\frac{1}{\tilde{\alpha}^4}}\right.\right.\right. \\
&\quad \left.\left.\left.\times\sum_{v\in V}\left(\frac{\Psi_{G_2}(j,v,w)\cdot l_{\tilde{d}-j,w}}{\Psi_{G_1}(\tilde{d},v,\tilde{\alpha})}\cdot\texttt{itv\_input}_v\right)-y_{j,w}\right)-\frac{1}{\tilde{\alpha}^6}\right)\right) \\
&\text{(by inductive hypothesis)} \\
&= \frac{\left(1-\frac{1}{\tilde{\alpha}^6}\right)^{j+1}}{1+\frac{1}{\tilde{\alpha}^4}}\sum_{v\in V}\left(\frac{l_{\tilde{d}-j-1,u}}{\Psi_{G_1}(\tilde{d},v,\tilde{\alpha})}\cdot\texttt{itv\_input}_v\cdot\sum_{w\in\Lambda_{G_4}(u)}\Psi_{G_2}(j,v,w)\right) \\
&\quad - \sum_{w\in\Lambda_{G_4}(u)}\left(\frac{l_{\tilde{d}-j-1,u}}{l_{\tilde{d}-j,w}}\left(\frac{1}{\tilde{\alpha}^6}+\left(1-\frac{1}{\tilde{\alpha}^6}\right)y_{j,w}\right)\right) \\
&\geq \frac{\left(1-\frac{1}{\tilde{\alpha}^6}\right)^{j+1}}{1+\frac{1}{\tilde{\alpha}^4}}\sum_{v\in V}\left(\frac{l_{\tilde{d}-j-1,u}}{\Psi_{G_1}(\tilde{d},v,\tilde{\alpha})}\cdot\texttt{itv\_input}_v\cdot\sum_{w\in\Lambda_{G_4}(u)}\Psi_{G_2}(j,v,w)\right) \\
&\quad - y_{j+1,u} \\
&\geq \frac{\left(1-\frac{1}{\tilde{\alpha}^6}\right)^{j+1}}{1+\frac{1}{\tilde{\alpha}^4}}\sum_{v\in V}\left(\frac{l_{\tilde{d}-j-1,u}}{\Psi_{G_1}(\tilde{d},v,\tilde{\alpha})}\cdot\texttt{itv\_input}_v\cdot\sum_{w\in\Lambda_{G_2}(u)}\Psi_{G_2}(j,v,w)\right) \\
&\quad - y_{j+1,u} \\
&= \frac{\left(1-\frac{1}{\tilde{\alpha}^6}\right)^{j+1}}{1+\frac{1}{\tilde{\alpha}^4}}\sum_{v\in V}\left(\frac{\Psi_{G_2}(j+1,v,u)\cdot l_{\tilde{d}-j-1,u}}{\Psi_{G_1}(\tilde{d},v,\tilde{\alpha})}\cdot\texttt{itv\_input}_v\right)-y_{j+1,u}
\end{aligned}$$

This completes our inductive proof for Eqs. (8) and (9).

Finally, Algorithm 2 sends messages only at Line 2 and Line 6. By Lemma 1, any node $u \in W$ always sends $O(\log n)$ bits in every round during the execution of Line 2. At Line 6, the algorithm sends a label AGGREGATE, a node id $\tilde{\alpha}$, $\texttt{round}^{-}(\texttt{remain}_{i,u},\tilde{\alpha})$, $l_{\tilde{d}-i+1,u}$, and $l_{\tilde{d}-i,u}$. We earlier proved that $\texttt{remain}_{i,u} \geq 0$

for all i and u, and that $\sum_{u \in W} \texttt{remain}_{i,u} = \sum_{u \in W} \texttt{itv\_input}_u$. This implies $0 \le \texttt{remain}_{i,u} \le \sum_{u \in W} \texttt{itv\_input}_u \le n + \tilde{\alpha}^{\tilde{d}} \le 2^{(n\tilde{\alpha})^4}$. Hence, by the property of $\texttt{round}^{-}()$, we can encode $\texttt{round}^{-}(\texttt{remain}_{i,u}, \tilde{\alpha})$ using $O(\log n)$ bits. Lemma 1 tells us that both $l_{\tilde{d}-i+1,u}$ and $l_{\tilde{d}-i,u}$ can also be encoded using $O(\log n)$ bits. Thus each message sent at Line 6 has just $O(\log n)$ bits. □

3.4 Building Block: Aggregating over Multiple Phases/Intervals

Following the ideas in Sect. 2.2 and Sect. 2.4, Algorithm 3 gives the subroutine for summing up all the `input` parameters of all the invoking nodes. This is done over multiple *phases*, with each phase consisting of $\tilde{d} \log \tilde{\alpha}$ *intervals*. At the beginning of each phase, each node has some value `remain`. In each interval in that phase, the node invokes `IntervalAggregate()` with $\frac{\texttt{remain}}{\tilde{d} \log \tilde{\alpha}}$ being the input. Each such invocation will end up with some leftover value. The sum of all the leftover values from all these intervals will be fed into the next phase. Note that under our invocation parameters later and when $T \ge cd^2 \log^2 n$, the total number of rounds in a phase will be no larger than T.

Later Algorithm 3 will be separately invoked for (i) counting the number of nodes, and (ii) counting the number of votes. For counting the number of votes, where the sum can be as large as n^d, Algorithm 3 will be invoked with $\texttt{max\_out} = \tilde{\alpha}^{\tilde{d}} = O(n^d)$ and $\texttt{reset} = \texttt{true}$. Having $\texttt{max\_out} = O(n^d)$ results in $O(d \log n)$ phases, which in turn ensures the leftover value to be less than 1 even if the sum is as large as n^d. Having $\texttt{reset} = \texttt{true}$ causes Algorithm 3 to invoke `ResetNeighbors()` after each phase, enabling the algorithm to fully utilize those newly created edges in the network (see Sect. 3.1). For counting the number of nodes, Algorithm 3 will be invoked with $\texttt{max\_out} = \tilde{\alpha} = O(n)$ and $\texttt{reset} = \texttt{false}$. This gives $O(\log n)$ phases, without `ResetNeighbors()` being invoked after each phase. Here we do not want to invoke `ResetNeighbors()`, so that later Algorithm 4 can properly distribute votes as desired.

The following lemma summarizes the guarantees of Algorithm 3. The lemma's proof largely follows the intuition in Sect. 2.2. For the lemma, recall that V denotes the set of all nodes in the network. Also, recall that for a node id x, we use "x" to refer to the integer value of x, and "node x" to refer to that node.

Lemma 3 (Guarantees of Algorithm 3) *Consider any round r, any node $\tilde{\alpha}$, any integer $\tilde{d}$ where $2 \le \tilde{d} \le \tilde{\alpha}$, any positive integer* `max_out`, *and any* `reset` $\in \{$***true***, ***false***$\}$. *Let W be any set of nodes where node $\tilde{\alpha} \in W$ and where each node $u \in W$ invokes* $\texttt{Aggregate}(\tilde{\alpha}, \tilde{d}, \texttt{input}_u, \texttt{max\_out}, \texttt{reset})$ *(i.e., Algorithm 3) in round r with some integer* $\texttt{input}_u \ge 0$ *such that* $\sum_{u \in W} \texttt{input}_u \le n + \tilde{\alpha}^{\tilde{d}}$. *Let* $\texttt{output}_{\tilde{\alpha}}$ *be the return value of Algorithm 3 on node $\tilde{\alpha}$, and let $r'' = r + 6\tilde{d}^2 \log \tilde{\alpha} \log(\texttt{max\_out})$. If no node in $V \setminus W$ interferes with instance $\tilde{\alpha}$ in any round from round r to round $r'' - 1$ (both inclusive), then we have:*

$$\texttt{output}_{\tilde{\alpha}} \le \sum_{u \in W} \texttt{input}_u \tag{10}$$

Algorithm 3 `Aggregate`.
/* Let z be the sum of all the `input` values on all the invoking nodes. */
Input: $\tilde{\alpha}$ (guess on α), $\tilde{d}$ (guess on d), `input` (integer), `max_out` (integer upper bound on z), and `reset` (whether `ResetNeighbors()` needs to be called at the end of every phase);
Output: Node $\tilde{\alpha}$ outputs z. We do not care about the outputs on other nodes.

```
1: procedure Aggregate($\tilde{\alpha}$, $\tilde{d}$, input, max_out, reset)
2:   remain ← input; ResetNeighbors();
3:   repeat $3\log($max_out$)$ times // This loop corresponds to $3\log($max_out$)$ phases. Each phase has $\tilde{d}\log\tilde{\alpha}$ intervals. Each interval has $2\tilde{d}$ rounds.
4:     if $\tilde{\alpha}$ = my id then itv_input ← 0; else itv_input ← remain/($\tilde{d}\log\tilde{\alpha}$);
5:     invoke IntervalAggregate($\tilde{\alpha}$, $\tilde{d}$, itv_input) for $\tilde{d}\log\tilde{\alpha}$ times (sequentially), let itv_output be the sum of the $\tilde{d}\log\tilde{\alpha}$ return values;
6:     if $\tilde{\alpha}$ = my id then remain ← remain + itv_output; else remain ← itv_output;
7:     if reset = true then ResetNeighbors();
8:   end
9:   return ⌈remain⌉;
```

We further have:

$$\texttt{output}_{\tilde{\alpha}} = \sum_{u\in W} \texttt{input}_u, \tag{11}$$

if all of the following three conditions hold:

- *$W = V$ and $\tilde{\alpha} \geq n$;*
- *$\sum_{u\in W} \texttt{input}_u \leq \texttt{max\_out}$;*
- *(`reset` $=$ **false** and $\tilde{d} \geq \Gamma_G(\tilde{\alpha})$, where $G = \sigma(r, r''))$ or (`reset` $=$ **true**, $\tilde{d} \geq d$, and $T \geq 3\tilde{d}^2 \log\tilde{\alpha}$).*

Finally, for any $u \in W$, node u always sends $O(\log n)$ bits in every round of its execution of Algorithm 3.

Proof We only prove the lemma for $n \geq 2$. The case for $n = 1$ is straightforward and much easier, and we omit for brevity. All line numbers below refer to Algorithm 3. We refer to Line 4 through 6 (both inclusive) as a phase. Let $\texttt{remain}_{0,u}$ (for all $u \in W$) be the value of `remain` on node u immediately after Line 2. For all $i \in [1, 3\log(\texttt{max\_out})]$ and all node $u \in W$, let $\texttt{remain}_{i,u}$, $\texttt{itv\_output}_{i,u}$, and $\texttt{itv\_input}_{i,u}$ be the value of the variables `remain`, `itv_output`, and `itv_input`, respectively, at the end of the i-th phase on node u. Consider any invocation of `IntervalAggregate()` at Line 5 in the i-th phase. Let round r'_i be the round when `ResetNeighbors()` was last invoked, before this invocation of `IntervalAggregate()`. One can easily verify from the pseudo-code that $r'_i = r$ if `reset` $=$ **false**, and $r'_i = r + (i-1)\cdot 2\tilde{d}^2\log\tilde{\alpha}$ if `reset` $=$ **true**.

For all i, we first prove (i) $\texttt{itv\_input}_{i,u} \geq 0$ for all $u \in W$ and (ii) $\sum_{u\in W} \texttt{itv\_input}_{i,u} \leq n + \tilde{\alpha}^{\tilde{d}}$, so that we can later invoke Lemma 2. We prove (i) and (ii) via an induction, together with two additional equations (iii) $\texttt{remain}_{i,u} \geq 0$ for all $u \in W$ and (iv) $\sum_{u\in W} \texttt{remain}_{i,u} = \sum_{u\in W} \texttt{input}_u$. For the base case of

$i = 1$, Equations (i) and (ii) obviously hold. Equations (iii) and (iv) follow from Eqs. (4) and (5) in Lemma 2. Next consider the inductive step of $i = j + 1$. For Equation (i), if $u = \tilde{\alpha}$, we trivially have $\texttt{itv\_input}_{j+1,u} \geq 0$. If $u \neq \tilde{\alpha}$, we have $\texttt{itv\_input}_{j+1,u} = \texttt{remain}_{j,u}/(\tilde{d} \log \tilde{\alpha}) \geq 0$. For Equation (ii), we have:

$$\begin{aligned}
\sum_{u \in W} \texttt{itv\_input}_{j+1,u} &= \sum_{u \in W \setminus \{\tilde{\alpha}\}} \frac{\texttt{remain}_{j,u}}{\tilde{d} \log \tilde{\alpha}} \\
&\leq \sum_{u \in W} \frac{\texttt{remain}_{j,u}}{\tilde{d} \log \tilde{\alpha}} \\
&= \sum_{u \in W} \frac{\texttt{input}_u}{\tilde{d} \log \tilde{\alpha}} \\
&\leq n + \tilde{\alpha}^{\tilde{d}}
\end{aligned}$$

For Equation (iii), if $u \neq \tilde{\alpha}$, by Lemma 2, we have $\texttt{remain}_{j+1,u} = \texttt{itv\_output}_{j+1,u} \geq 0$. If $u = \tilde{\alpha}$, by Lemma 2 and by the inductive hypothesis, we have:

$$\begin{aligned}
\texttt{remain}_{j+1,u} &= \texttt{remain}_{j,\tilde{\alpha}} + \texttt{itv\_output}_{j+1,u} \\
&\geq \texttt{itv\_output}_{j+1,u} \geq 0
\end{aligned}$$

Finally for Equation (iv), we have:

$$\begin{aligned}
\sum_{u \in W} \texttt{remain}_{j+1,u} &= (\texttt{remain}_{j,\tilde{\alpha}} + \texttt{itv\_output}_{j+1,\tilde{\alpha}}) \\
&\quad + \sum_{u \in W \setminus \{\tilde{\alpha}\}} \texttt{itv\_output}_{j+1,u} \\
&= \texttt{remain}_{j,\tilde{\alpha}} + \sum_{u \in W} \texttt{itv\_output}_{j+1,u} \\
&= \text{(by Lemma 2)}\ \texttt{remain}_{j,\tilde{\alpha}} + \tilde{d} \log \tilde{\alpha} \sum_{u \in W} \texttt{itv\_input}_{j+1,u} \\
&= \texttt{remain}_{j,\tilde{\alpha}} + 0 + \tilde{d} \log \tilde{\alpha} \sum_{u \in W \setminus \{\tilde{\alpha}\}} \texttt{itv\_input}_{j+1,u} \\
&= \texttt{remain}_{j,\tilde{\alpha}} + \sum_{u \in W \setminus \{\tilde{\alpha}\}} \texttt{remain}_{j,u} \\
&= \sum_{u \in W} \texttt{remain}_{j,u} \\
&= \text{(by inductive hypothesis)} \sum_{u \in W} \texttt{input}_u
\end{aligned}$$

This completes the inductive proof for the four equations.

Define $z = \sum_{u \in W} \texttt{input}_u$ and $x = 3\log(\texttt{max\_out})$. We can now prove Eq. (10):

$$\begin{aligned}
\texttt{output}_{\tilde{\alpha}} &= \lceil \texttt{remain}_{x,\tilde{\alpha}} \rceil \\
&\leq \text{ (by Equation (iii))} \left\lceil \sum_{u \in W} \texttt{remain}_{x,u} \right\rceil \\
&= \text{(by Equation (iv))} \left\lceil \sum_{u \in W} \texttt{input}_u \right\rceil \\
&= \sum_{u \in W} \texttt{input}_u \text{(since } \texttt{input}_u \text{ is an integer for all } u\text{)}
\end{aligned}$$

We next move on to Eq. (11). Let $G_i = \sigma(r'_i, r + i \cdot 2\tilde{d}^2 \log \tilde{\alpha})$ for all $i \in [1, x]$. We first claim that $\tilde{d} \geq \Gamma_{G_i}(\tilde{\alpha})$ for all i: First, consider the case where $\texttt{reset} =$ **false**. We then have $r'_i = r$. This means that $\sigma(r, r'')$ must be a subgraph of G_i. Hence by the condition $\tilde{d} \geq \Gamma_G(\tilde{\alpha})$ in the lemma, we must have $\tilde{d} \geq \Gamma_{G_i}(\tilde{\alpha})$. Second, if $\texttt{reset} =$ **true**, then $r'_i = r+(i-1)\cdot 2\tilde{d}^2 \log \tilde{\alpha}$ and $G_i = \sigma(r+(i-1)\cdot 2\tilde{d}^2 \log \tilde{\alpha}, r+i\cdot 2\tilde{d}^2 \log \tilde{\alpha})$. By the condition $T \geq 3\tilde{d}^2 \log \tilde{\alpha}$ in the lemma, and also by the definition of backbone diameter of T-interval dynamic networks, we know that the diameter of (the graph) G_i is at most d. By the condition of $\tilde{d} \geq d$ in the lemma, we have $\tilde{d} \geq \Gamma_{G_i}(\tilde{\alpha})$.

Next, for all $i \in [0, x]$, define $y_i = \sum_{u \in V} \texttt{remain}_{i,u} - \texttt{remain}_{i,\tilde{\alpha}}$. We will later prove that if $W = V$, $\tilde{\alpha} \geq n$, and $\tilde{d} \geq \Gamma_{G_i}(\tilde{\alpha})$ for all $i \in [1, x]$, then $y_i \leq \frac{5}{8} y_{i-1} + \frac{n}{\tilde{\alpha}^3}$ for all $i \in [1, x]$. This will imply that if we further have $z \leq \texttt{max\_out}$, then:

$$\begin{aligned}
\texttt{output}_{\tilde{\alpha}} &= \lceil \texttt{remain}_{x,\tilde{\alpha}} \rceil \\
&= \left\lceil \sum_{u \in V} \texttt{remain}_{x,u} - y_x \right\rceil \\
&= \text{ (by Equation (iv)) } \lceil z - y_x \rceil \\
&\geq \left\lceil z - \left(\left(\frac{5}{8}\right)^x y_0 + \frac{n}{\tilde{\alpha}^3} \sum_{i=0}^{x-1} \left(\frac{5}{8}\right)^i \right) \right\rceil \\
&\geq \left\lceil z - \left(\frac{1}{4\texttt{max\_out}} y_0 + \frac{8n}{3\tilde{\alpha}^3} \right) \right\rceil \\
&\geq \left\lceil z - \frac{1}{4\texttt{max\_out}} z - \frac{8n}{3\tilde{\alpha}^3} \right\rceil \\
&\geq \left\lceil z - \frac{1}{4} - \frac{8}{3n^2} \right\rceil \\
&= z \left(\text{since } z = \sum_{u \in W} \texttt{input}_u \text{ and must be an integer} \right)
\end{aligned}$$

This proves Eq. (11).

The following proves that if $W = V$, $\tilde{\alpha} \geq n$, and $\tilde{d} \geq \Gamma_{G_i}(\tilde{\alpha})$ for all $i \in [1, x]$, then $y_i \leq \frac{5}{8} y_{i-1} + \frac{n}{\tilde{\alpha}^3}$ for all $i \in [1, x]$. By Equation (iv), we have:

$$\sum_{u \in V} \texttt{remain}_{i,u} = z = \sum_{u \in V} \texttt{remain}_{i-1,u}$$

Thus:

$$\begin{aligned} y_i &= \sum_{u \in V} \texttt{remain}_{i,u} - \texttt{remain}_{i,\tilde{\alpha}} \\ &= \sum_{u \in V} \texttt{remain}_{i-1,u} - (\texttt{remain}_{i-1,\tilde{\alpha}} + \texttt{itv\_output}_{i,\tilde{\alpha}}) \\ &= y_{i-1} - \texttt{itv\_output}_{i,\tilde{\alpha}} \end{aligned}$$

It suffices to prove $\texttt{itv\_output}_{i,\tilde{\alpha}} \geq \frac{3}{8} y_{i-1} - \frac{n}{\tilde{\alpha}^3}$. Consider any given i. For all $j \in [1, \tilde{d} \log \tilde{\alpha} + 1]$, let $G_{i,j} = \sigma(r'_i, r + (i-1)2\tilde{d}^2 \log \tilde{\alpha} + (j-1)2\tilde{d})$. Note that G_i is a subgraph of $G_{i,j}$ for all j, hence $\tilde{d} \geq \Gamma_{G_i}(\tilde{\alpha}) \geq \Gamma_{G_{i,j}}(\tilde{\alpha})$. We thus have:

$$\begin{aligned} &\texttt{itv\_output}_{i,\tilde{\alpha}} \\ &\geq \sum_{j \in [1, \tilde{d} \log \tilde{\alpha}]} \left(\frac{3}{4} \sum_{u \in V} \left(\frac{\Psi_{G_{i,j+1}}(\tilde{d}, u, \tilde{\alpha})}{\Psi_{G_{i,j}}(\tilde{d}, u, \tilde{\alpha})} \cdot \texttt{itv\_input}_{i,u} \right) - \frac{n}{\tilde{\alpha}^5} \right) \text{ (by Lemma 2)} \\ &= -\frac{n \cdot \tilde{d} \log \tilde{\alpha}}{\tilde{\alpha}^5} + \frac{3}{4} \sum_{u \in V} \left(\texttt{itv\_input}_{i,u} \cdot \sum_{j \in [1, \tilde{d} \log \tilde{\alpha}]} \frac{\Psi_{G_{i,j+1}}(\tilde{d}, u, \tilde{\alpha})}{\Psi_{G_{i,j}}(\tilde{d}, u, \tilde{\alpha})} \right) \\ &\geq -\frac{n}{\tilde{\alpha}^3} + \frac{3}{4} \sum_{u \in V} \left(\texttt{itv\_input}_{i,u} \cdot \left(\prod_{j \in [1, \tilde{d} \log \tilde{\alpha}]} \frac{\Psi_{G_{i,j+1}}(\tilde{d}, u, \tilde{\alpha})}{\Psi_{G_{i,j}}(\tilde{d}, u, \tilde{\alpha})} \right)^{\frac{1}{\tilde{d} \log \tilde{\alpha}}} \cdot \tilde{d} \log \tilde{\alpha} \right) \\ &\geq -\frac{n}{\tilde{\alpha}^3} + \frac{3}{4} \sum_{u \in V} \left(\texttt{itv\_input}_{i,u} \cdot \left(\frac{\Psi_{G_{i,\tilde{d} \log \tilde{\alpha}+1}}(\tilde{d}, u, \tilde{\alpha})}{\Psi_{G_{i,1}}(\tilde{d}, u, \tilde{\alpha})} \right)^{\frac{1}{\tilde{d} \log \tilde{\alpha}}} \cdot \tilde{d} \log \tilde{\alpha} \right) \\ &\geq -\frac{n}{\tilde{\alpha}^3} + \frac{3}{4} \sum_{u \in V} \left(\texttt{itv\_input}_{i,u} \cdot \left(\frac{1}{n^{\tilde{d}}} \right)^{\frac{1}{\tilde{d} \log \tilde{\alpha}}} \cdot \tilde{d} \log \tilde{\alpha} \right) \\ &\quad \text{(the above holds because } \Psi_{G_{i,1}}(\tilde{d}, u, \tilde{\alpha}) \leq n^{\tilde{d}}, \Gamma_{G_{i,\tilde{d} \log \tilde{\alpha}+1}}(\tilde{\alpha}) \leq \tilde{d}, \\ &\quad \text{and } \Psi_{G_{i,\tilde{d} \log \tilde{\alpha}+1}}(\tilde{d}, u, \tilde{\alpha}) \geq 1) \\ &\geq -\frac{n}{\tilde{\alpha}^3} + \frac{3}{4} \sum_{u \in V} \left(\texttt{itv\_input}_{i,u} \cdot \left(\frac{1}{\tilde{\alpha}^{\tilde{d}}} \right)^{\frac{1}{\tilde{d} \log \tilde{\alpha}}} \cdot \tilde{d} \log \tilde{\alpha} \right) \end{aligned}$$

$$\geq -\frac{n}{\tilde{\alpha}^3} + \frac{3}{8}\sum_{u\in V}\left(\texttt{itv\_input}_{i,u}\cdot \tilde{d}\log\tilde{\alpha}\right)$$

$$= -\frac{n}{\tilde{\alpha}^3} + \frac{3}{8}\left(\sum_{u\in V}\texttt{remain}_{i-1,u} - \texttt{remain}_{i-1,\tilde{\alpha}}\right) = -\frac{n}{\tilde{\alpha}^3} + \frac{3}{8}y_{i-1}$$

This completes our proof that if $W = V$, $\tilde{\alpha} \geq n$, and $\tilde{d} \geq \Gamma_{G_i}(\tilde{\alpha})$ for all $i \in [1, x]$, then $y_i \leq \frac{5}{8}y_{i-1} + \frac{n}{\tilde{\alpha}^3}$ for all $i \in [1, x]$.

Finally, the claim on the number of bits sent in each round directly follows from Lemma 2. □

3.5 Building Block: Distributing Votes

Following the ideas in Sect. 2.4, in order to check whether $\tilde{d}$ is sufficiently large, node $\tilde{\alpha}$ will distribute $\tilde{\alpha}^{\tilde{d}}$ votes to all the nodes, over about $\tilde{d}$ rounds. (By the definition of α and by the discussion in Sect. 2.4, we must have $\tilde{\alpha} \leq \alpha \leq \text{poly}(n)$.) The quantity $\tilde{\alpha}^{\tilde{d}}$ ensures that each node, within distance of $\tilde{d}$ from node $\tilde{\alpha}$, gets at least one vote when $\tilde{\alpha} \geq n$. Algorithm 4 gives the subroutine for doing this. A simple design would be for each node, upon receiving some votes for the first time, to keep one vote for itself and distribute all the remaining votes to its neighbors. But this would need $\Theta(\tilde{d}\log\tilde{\alpha})$ message size. To reduce the message size to $O(\log n)$, in each round each node in Algorithm 4 only sends a single bit (in addition to also sending $\tilde{\alpha}$) indicating whether it has any votes. In the i-th iteration, if node u has votes while a neighboring node v has none, then exactly $\tilde{\alpha}^{\tilde{d}-i}$ votes are transferred from node u to node v. Here the quantity $\tilde{\alpha}^{\tilde{d}-i}$ is implicit. We will show that when doing so, a node never runs out of votes to distribute, as long as it never has more than $\tilde{\alpha}$ neighbors. On the other hand, if the number of neighbors exceeds $\tilde{\alpha}$ (at Line 4), in Algorithm 4 the node simply refuses to distribute any vote, and will cause vote verification later to fail. This is intentional since more than $\tilde{\alpha}$ neighbors implies $\tilde{\alpha} < n$ and hence $\tilde{\alpha} \neq \alpha$. Failing the vote verification then forces the algorithm to later update $\tilde{\alpha}$. The following lemma summarizes the guarantees of Algorithm 4. (Also, recall that V denotes the set of all nodes in the network.)

Lemma 4 (Guarantees of Algorithm 4) *Consider any rounds r' and r where $r' \leq r$, any node $\tilde{\alpha}$, and any integer $\tilde{d}$ where $2 \leq \tilde{d} \leq \tilde{\alpha}$. Let W be any set of nodes that all (i) invoke* $\texttt{DistributeVotes}(\tilde{\alpha}, \tilde{d})$ *(i.e., Algorithm 4) simultaneously in round r, and (ii) last invoked* $\texttt{ResetNeighbors}()$ *at the beginning of round r'. Let $G_1 = \sigma(r', r)$ and $G_2 = \sigma(r', r+\tilde{d}+2)$. For $u \in W$, let* $\texttt{output}_u$ *be the return value of Algorithm 4 on node u. If (i) $\tilde{\alpha} \in W$, and (ii) for all $v \in V \setminus W$ and in every round from round r to round $r + \tilde{d} + 1$ (both inclusive), node v sends some message but does not interfere with instance $\tilde{\alpha}$, then all the following holds:*

- $\sum_{u\in W}\texttt{output}_u \leq \tilde{\alpha}^{\tilde{d}}$ *and* $\texttt{output}_u$ *is a non-negative integer for all $u \in W$.*
- *If* $\sum_{u\in W}\texttt{output}_u = \tilde{\alpha}^{\tilde{d}}$*, then $W = V$ and $\tilde{d} \geq \Gamma_{G_1}(\tilde{\alpha})$.*
- *If $W = V$, $\tilde{d} \geq \Gamma_{G_2}(\tilde{\alpha})$, and $\tilde{\alpha} \geq n$, then* $\sum_{u\in W}\texttt{output}_u = \tilde{\alpha}^{\tilde{d}}$.

Algorithm 4 `DistributeVotes`.
/* This subroutine distributes $\tilde{\alpha}^{\tilde{d}}$ votes.*/
Input: $\tilde{\alpha}$ (guess on α) and $\tilde{d}$ (guess on d);
Output: Each node outputs the number of votes it ends up having.

```
1: procedure DistributeVotes($\tilde{\alpha}, \tilde{d}$)
2:     if $\tilde{\alpha}$ = my id then votes ← $\tilde{\alpha}^{\tilde{d}}$; else votes ← 0;
3:     send ⟨NOTIFY, $\tilde{\alpha}$⟩;                                    ▷ Takes one round.
4:     if I receive more than $\tilde{\alpha}$ messages of the form ⟨NOTIFY, $\tilde{\alpha}'$⟩ with $\tilde{\alpha}' = \tilde{\alpha}$ then bad ← true; else
       bad ← false;
5:     for $i$ ← 1, ..., $\tilde{d}$ do                                   ▷ This loop takes total $\tilde{d}$ rounds.
6:         if (votes > 0) and (bad = false) then
7:             send ⟨HAS_VOTE, $\tilde{\alpha}$⟩, then let $x_1$ be the number of ⟨NO_VOTE, $\tilde{\alpha}'$⟩ messages received where
               $\tilde{\alpha}' = \tilde{\alpha}$;
8:             votes ← votes − $x_1 \tilde{\alpha}^{\tilde{d}-i}$;
9:         else
10:            send ⟨NO_VOTE, $\tilde{\alpha}$⟩, then let $x_2$ be the number of ⟨HAS_VOTE, $\tilde{\alpha}'$⟩ messages received where
               $\tilde{\alpha}' = \tilde{\alpha}$;
11:            votes ← votes + $x_2 \tilde{\alpha}^{\tilde{d}-i}$;
12:        end if
13:    end for
14:    if votes = 0 then send ⟨FAIL, $\tilde{\alpha}$⟩; else send ⟨NO_FAIL, $\tilde{\alpha}$⟩;        ▷ Takes one round.
15:    if there exists some node such that in this round: i) I do not receive ⟨NO_FAIL, $\tilde{\alpha}'$⟩ with $\tilde{\alpha}' = \tilde{\alpha}$ from
       that node, and ii) I receive some (other) message from that node then return 0; else return votes; /*
       For this line, a node takes into account both oldcomer messages and newcomers messages. */
```

Finally, for all $u \in W$, node u always sends $O(\log n)$ bits in every round of its execution of Algorithm 4.

Proof All line numbers below refer to Algorithm 4. The claim on the number of bits sent in each round is obvious from the pseudo-code of Algorithm 4. Let $\texttt{votes}_u$ be the `votes` variable in Algorithm 4 on node $u \in W$. Consider the iteration from Line 6 to 12 (both inclusive). We first claim that one of the following two cases must hold:

Case 1: $\texttt{votes}_u = 0$ in all rounds in all $\tilde{d}$ iterations.
Case 2: $\texttt{votes}_u = 0$ in all rounds in iterations 1 through $i-1$ for some $i \in [1, \tilde{d}]$, and $\texttt{votes}_u$ is a positive integer in all rounds in iterations i through $\tilde{d}$.

We first prove the above claim for $u \neq \tilde{\alpha}$. Let iteration i be the first iteration during which $\texttt{votes}_u$ gets assigned a non-zero value. If such i does not exist, then Case 1 holds. Otherwise by the condition at Line 6 and also by Line 11, we have $\texttt{votes}_u \geq \tilde{\alpha}^{\tilde{d}-i} > 0$ at the end of iteration i. In each future iteration j where $i+1 \leq j \leq \tilde{d}$, $\texttt{votes}_u$ can decrease by at most $(\tilde{\alpha}-1) \cdot \tilde{\alpha}^{\tilde{d}-j}$. Since $\tilde{\alpha}^{\tilde{d}-i} - (\tilde{\alpha}-1) \cdot \tilde{\alpha}^{\tilde{d}-i-1} - (\tilde{\alpha}-1) \cdot \tilde{\alpha}^{\tilde{d}-i-2} - \cdots - (\tilde{\alpha}-1) \cdot \tilde{\alpha}^0 \geq 1$, we have $\texttt{votes}_u > 0$ in all rounds in iterations i through $\tilde{d}$. One can further trivially verify that $\texttt{votes}_u$ must always be an integer. We have thus finished proving the above claim for $u \neq \tilde{\alpha}$. For $u = \tilde{\alpha}$, one can easily prove (in a similar way) that Case 2 always holds.

We next show that $\sum_{u \in W} \texttt{votes}_u = \tilde{\alpha}^{\tilde{d}}$ immediately after Line 13. We obviously have $\sum_{u \in W} \texttt{votes}_u = \tilde{\alpha}^{\tilde{d}}$ immediately after Line 2. The value of `votes` on a node can only change at Lines 8 and 11. Consider any given iteration from Line 6 to Line 12, which has only a single round. Consider any node $u \in W$ and any node v that is a neighbor of node u in that round. If $v \notin W$, then by the condition in the lemma, node v does not interfere with instance $\tilde{\alpha}$ in that round. By the pseudocode, the message from node v to node u will be ignored by the algorithm, and has no effect on $\texttt{votes}_u$ and $\sum_{u \in W} \texttt{votes}_u$. Next consider the case where $v \in W$. If node u sends $\langle \texttt{HAS\_VOTE}, \tilde{\alpha} \rangle$ and node v sends $\langle \texttt{NO\_VOTE}, \tilde{\alpha} \rangle$, then such message exchanged between node u and node v will decrease (increase) $\texttt{votes}_u$ ($\texttt{votes}_v$) by $\tilde{\alpha}^{i-1}$. This means that $\sum_{u \in W} \texttt{votes}_u$ does not change due to this message exchanged between node u and node v. One can verify that the same holds in all the remaining 3 cases (e.g., when both node u and node v send $\langle \texttt{HAS\_VOTE}, \tilde{\alpha} \rangle$). Putting everything together, we know that $\sum_{u \in W} \texttt{votes}_u = \tilde{\alpha}^{\tilde{d}}$ immediately after Line 13.

We are now ready to prove the three properties claimed by the lemma:

- The first property directly follows from (i) $\texttt{votes}_u$ must always be a non-negative integer, (ii) $\sum_{u \in W} \texttt{votes}_u = \tilde{\alpha}^{\tilde{d}}$ immediately after Line 13, and (iii) the pseudocode at Line 15.
- We prove the second property by showing that if $W \neq V$ or $\tilde{d} < \Gamma_{G_1}(\tilde{\alpha})$, then $\sum_{u \in W} \texttt{output}_u < \tilde{\alpha}^{\tilde{d}}$. Let W' be the set of nodes u in W such that $\texttt{votes}_u > 0$ immediately after Line 13. Note that we have $\texttt{votes}_{\tilde{\alpha}} > 0$ in the first round of the algorithm, and hence $\texttt{votes}_{\tilde{\alpha}}$ must satisfy Case 2 with $i = 1$. This means that $\tilde{\alpha} \in W'$ and hence, W' is non-empty.
  Next since $W \neq V$ or $\tilde{d} < \Gamma_{G_1}(\tilde{\alpha})$, we must have $W' \neq V$. Consider the topology of the dynamic network at Line 14 (i.e., in round $r + \tilde{d} + 1$). Since the topology in each round is always connected and since W' is non-empty, there must exist neighboring nodes u and v such that $u \in W'$ and $v \in V \setminus W'$. If $v \in W$, then node v will satisfy the condition at Line 14 and send $\langle \texttt{FAIL}, \tilde{\alpha} \rangle$ in round $r + \tilde{d} + 1$. If $v \notin W$, then by the condition in the lemma, node v will send some message but does not interfere with instance $\tilde{\alpha}$ in that round. In both cases the condition at Line 15 will be satisfied on node u, causing node u to return 0 instead of $\texttt{votes}_u$. (Recall that at Line 15 a node uses both oldcomer and newcomer messages.) We showed earlier that $\sum_u \texttt{votes}_{u \in W} = \tilde{\alpha}^{\tilde{d}}$ immediately after Line 13. Hence we have $\sum_{u \in W} \texttt{output}_u < \tilde{\alpha}^{\tilde{d}}$.
- We move on to prove the third property claimed by the lemma. If $\tilde{\alpha} \geq n$, then obviously each node has at most $\tilde{\alpha}$ neighbors, and $\texttt{bad}$ will be set to **false** at Line 4 (and remain **false** throughout). We first prove that for any node $u \in W$, there exists i ($1 \leq i \leq j$) such that $\texttt{votes}_u > 0$ immediately after Line 12 of the i-th iteration. Here j is the length (in terms of the number of hops) of the shortest path from node u to node $\tilde{\alpha}$ in G_2. We use a simple induction on j. The induction base for $j = 1$ is trivial. For the inductive step, suppose that the hypothesis holds for $j = k$. Consider any node u whose shortest path to node $\tilde{\alpha}$ in G_2 has a length of $k + 1$, and let node v be the node immediately after node u on any such path. Then by the inductive hypothesis there exists some i ($1 \leq i \leq k$) such that $\texttt{votes}_v > 0$ immediately after Line 12 of the i-th iteration. Then in the $(i+1)$-th iteration, node v will send $\langle \texttt{HAS\_VOTE}, \tilde{\alpha} \rangle$. If $\texttt{votes}_u = 0$ at Line 6 in the $(i + 1)$-th iteration, then node u will execute Line 11, which will make $\texttt{votes}_u > 0$ immediately

after Line 12 of the $(i+1)$-th iteration. Otherwise if $\texttt{votes}_u > 0$ at Line 6 in the $(i+1)$-th iteration, then $\texttt{votes}_u$ must belong to Case 2, and $\texttt{votes}_u$ will remain positive immediately after Line 12 of the $(i+1)$-th iteration.
Finally, we showed earlier (in Case 2) that if $\texttt{votes}_u > 0$ immediately after Line 12 of the i-th iteration, then $\texttt{votes}_u > 0$ in all future iterations. Since $W = V$ and $\tilde{d} \geq \Gamma_{G_2}(\tilde{\alpha})$, at the end of the $\tilde{d}$-th iteration we must have $\texttt{votes}_u > 0$ on all nodes. Hence, all nodes will send $\langle \texttt{NO\_FAIL}, \tilde{\alpha} \rangle$ at Line 14. Together with our earlier claim that $\sum_u \texttt{votes}_{u \in W} = \tilde{\alpha}^{\tilde{d}}$ immediately after Line 13, this implies $\sum_{u \in W} \texttt{output}_u = \tilde{\alpha}^{\tilde{d}}$.

□

3.6 Putting Everything Together

Algorithm 5 gives our final algorithm for COUNT, which follows the intuition in Sect. 2.4. Namely, the nodes first invoke Aggregate() to count the number of nodes. Next they use DistributeVotes() to distribute $\tilde{\alpha}^{\tilde{d}}$ votes, and then invoke Aggregate() again to count the total number of votes. If the second Aggregate() result matches $\tilde{\alpha}^{\tilde{d}}$, the nodes output.

In Algorithm 5, each node runs CountNodes() and FloodRoot() in parallel. In FloodRoot(), each node keeps sending the largest id that it has seen so far. If this id equals its own id, then the node will initiate a new "instance", by flooding a SYNC message. Multiple nodes may start flooding such SYNC messages simultaneously. Other nodes will wait until they receive the first such SYNC message, and join the "instance" corresponding to that SYNC message. The SYNC message also carries information on when this "instance" should start—this enables all nodes in the "instance" to invoke Aggregate() and DistributeVotes() in a synchronized fashion (at Line 14–16). A node may need to switch from one "instance" to another. To avoid various technical issues, we do not allow such switch to happen during the invocations of Aggregate() and DistributeVotes(), and hence the switch may be delayed. Finally, all the SWITCH/SYNC/OUTPUT messages in the algorithm are "signaling" messages, and we want nodes to process them as soon as possible. Hence a node will process (rather than ignore) these messages even if they are newcomer messages.

With FloodRoot(), all nodes will see the largest id α (among the n nodes) within d rounds. All nodes hence will later switch into the "instance" whose root is node α, which eventually causes the algorithm to produce a correct answer. A careful reasoning will show that the delayed switches will not blow up the time complexity. The following theorem summarizes the final guarantees of our COUNT algorithm:

Theorem 5 *There exists some constant c independent of d, n, and T, such that as long as $T \geq cd^2 \log^2 n$:*

- *Algorithm 5 always outputs n (and terminates) in $O(d^3 \log^2 n)$ rounds.*
- *In each round during the execution of Algorithm 5, each node sends only $O(\log n)$ bits.*

Algorithm 5 `CountNodes` and `FloodRoot`.
/* For counting the number of nodes.
Here `CountNodes`() and `FloodRoot`() should be invoked concurrently, and should run in parallel. In this algorithm, all newcomer messages in the form of $\langle x, \ldots \rangle$ where $x \in \{$`SWITCH`, `SYNC`, `OUTPUT`$\}$ will be used instead of being ignored. */
Input: Nothing;
Output: n

```
1: procedure CountNodes()
2:     $\tilde{d} \leftarrow 1$;
3:     repeat forever
4:         tmp ← largest $\tilde{\alpha}'$ among all the $\langle$SWITCH, $\tilde{\alpha}'\rangle$ messages that I have ever received (if no such
   message have been received, then tmp ← my id);
5:         if tmp = my id then
6:             $\tilde{\alpha} \leftarrow$ tmp; $\tilde{d} \leftarrow \min(2\tilde{d}, \tilde{\alpha})$; num_round ← $\tilde{d}$;
7:         else
8:             wait until I receive some $\langle$SYNC, $\tilde{\alpha}', \tilde{d}'$, num_round'$\rangle$ message;
9:             $\tilde{\alpha} \leftarrow \tilde{\alpha}'$; $\tilde{d} \leftarrow \tilde{d}'$; num_round ← num_round';
10:        end if

11:        while num_round > 0 do                              ▷ This loop takes at most $\tilde{d}$ rounds.
12:            num_round ← num_round − 1; send $\langle$SYNC, $\tilde{\alpha}, \tilde{d}$, num_round$\rangle$;
13:        end while

14:        result ← Aggregate($\tilde{\alpha}, \tilde{d}, 1, \tilde{\alpha}$, false);          ▷ Takes total $6\tilde{d}^2 \log^2 \tilde{\alpha}$ rounds.
15:        votes ← DistributeVotes($\tilde{\alpha}, \tilde{d}$);                    ▷ Takes total $2 + \tilde{d}$ rounds.
16:        collected ← Aggregate($\tilde{\alpha}, \tilde{d}$, votes, $\tilde{\alpha}^{\tilde{d}}$, true);    ▷ Takes total $6\tilde{d}^3 \log^2 \tilde{\alpha}$ rounds.

17:        if ($\tilde{\alpha}$ = my id and collected = $\tilde{\alpha}^{\tilde{d}}$) then send $\langle$OUTPUT, result$\rangle$, output result, and
   terminate;
18:        /* Upon receiving $\langle$OUTPUT, result$\rangle$, in the next round, a node sends $\langle$OUTPUT, result$\rangle$,
   outputs result, and terminates. */
19:    end

20: procedure FloodRoot()
21:    tmp ← my id;
22:    repeat forever
23:        send $\langle$SWITCH, tmp$\rangle$; tmp ← largest $\tilde{\alpha}'$ among all the $\langle$SWITCH, $\tilde{\alpha}'\rangle$ messages I have ever
   received;
24:    end
```

3.7 Proof for Theorem 5

This section proves Theorem 5. To facilitate the proof, we define the notion of *subexecution*. For any round r, any integer $\tilde{d}$, and any nodes u and $\tilde{\alpha}$, we say that *subexecution*$(u, r, \tilde{\alpha}, \tilde{d})$ *exists* if during node u's execution of Algorithm 5, node u invokes `Aggregate`($\tilde{\alpha}, \tilde{d}, 1, \tilde{\alpha}$, **false**) at Line 14 in round r. If subexecution$(u, r, \tilde{\alpha}, \tilde{d})$ does exist, then we use subexecution$(u, r, \tilde{\alpha}, \tilde{d})$ to refer to node u's execution of Line 11 through 16 (both inclusive), with Line 14 being invoked in round r. Note that the values of $\tilde{\alpha}$ and $\tilde{d}$ on node u never change during a subexecution.

In the following, we first prove Lemmas 6 and 7, and then prove Theorem 5.

Lemma 6 *Consider any round r, any node $\tilde{\alpha}$, and any integer $\tilde{d}$ where $2 \le \tilde{d} \le \tilde{\alpha}$. Let $W = \{u \mid \textit{subexecution}(u, r, \tilde{\alpha}, \tilde{d})\ \textit{exists}\}$. If $W \ne \emptyset$, then (i) $\tilde{\alpha} \in W$, and (ii) for all $v \in V \setminus W$ and $r' \in [r, r + 1 + \tilde{d} + 6\tilde{d}^2 \log^2 \tilde{\alpha} + 6\tilde{d}^3 \log^2 \tilde{\alpha}]$, in round r' of the execution of Algorithm 5, node v does not interfere with instance $\tilde{\alpha}$.*

Proof All line numbers below refer to Algorithm 5. Consider any W where $W \ne \emptyset$. We first prove $\tilde{\alpha} \in W$. Let u be any node in W. If $u = \tilde{\alpha}$, then we are done. Otherwise node u must have received $\langle \texttt{SYNC}, \tilde{\alpha}', \tilde{d}', \texttt{num\_round}' \rangle$ at Line 8 with $\tilde{\alpha}' = \tilde{\alpha}$ and $\tilde{d}' = \tilde{d}$ in round $r - \texttt{num\_round}' - 1$. From the pseudo-code, one can then easily verify that node $\tilde{\alpha}$ must have sent $\langle \texttt{SYNC}, \tilde{\alpha}, \tilde{d}, \tilde{d} - 1 \rangle$ in round $r - \tilde{d}$. Hence node $\tilde{\alpha}$ will execute Line 14 starting at round $r - \tilde{d} + \tilde{d} = r$, which means that subexecution$(\tilde{\alpha}, r, \tilde{\alpha}, \tilde{d})$ must exist and that $\tilde{\alpha} \in W$.

Next, consider any node $v \in V \backslash W$ and any $r' \in [r, r + 1 + \tilde{d} + 6\tilde{d}^2 \log^2 \tilde{\alpha} + 6\tilde{d}^3 \log^2 \tilde{\alpha}]$. Obviously, we only need to prove that if node v has not terminated in round r', then node v does not interfere with instance $\tilde{\alpha}$ in round r'. We enumerate all possible places where node v can send a message in round r': (i) The messages sent at Lines 17 and 23 will never cause node v to interfere with instance $\tilde{\alpha}$. (ii) If v sends a message somewhere between Lines 11 and 16 in round r', then node v in round r' must be in the middle of some subexecution$(v, r_0, \tilde{\alpha}_0, \tilde{d}_0)$ for some r_0, $\tilde{\alpha}_0$, and $\tilde{d}_0$. It suffices to prove that $\tilde{\alpha}_0 \ne \tilde{\alpha}$. We prove by contradiction and assume $\tilde{\alpha}_0 = \tilde{\alpha}$. Define $X = \{x \mid \text{subexecution}(x, r_0, \tilde{\alpha}, \tilde{d}_0) \text{ exists}\}$, and hence $v \in X$ and $X \ne \emptyset$. By the first part of the lemma, we know that $\tilde{\alpha} \in X$ and that subexecution$(\tilde{\alpha}, r_0, \tilde{\alpha}, \tilde{d}_0)$ exists. Furthermore in round r', since node v is in the middle of subexecution$(v, r_0, \tilde{\alpha}, \tilde{d}_0)$, node $\tilde{\alpha}$ must also be in the middle of subexecution$(\tilde{\alpha}, r_0, \tilde{\alpha}, \tilde{d}_0)$ (i.e. the subexecution has not ended).

Recall that we earlier already showed that subexecution$(\tilde{\alpha}, r, \tilde{\alpha}, \tilde{d})$ also exists. Since $r' \in [r, r + 1 + \tilde{d} + 6\tilde{d}^2 \log^2 \tilde{\alpha} + 6\tilde{d}^3 \log^2 \tilde{\alpha}]$, in round r' node $\tilde{\alpha}$ must still be in the middle of subexecution$(\tilde{\alpha}, r, \tilde{\alpha}, \tilde{d})$ (i.e. the subexecution has not ended). At any given point of time, node $\tilde{\alpha}$ can only be in a single subexecution. Hence the two subexecutions, subexecution$(\tilde{\alpha}, r_0, \tilde{\alpha}, \tilde{d}_0)$ and subexecution$(\tilde{\alpha}, r, \tilde{\alpha}, \tilde{d})$, must be the same, and we have $\tilde{d}_0 = \tilde{d}$ and $r_0 = r$. This then means that subexecution$(v, r_0, \tilde{\alpha}_0, \tilde{d}_0)$ is the same as subexecution$(v, r, \tilde{\alpha}, \tilde{d})$, and that subexecution$(v, r, \tilde{\alpha}, \tilde{d})$ exists. This implies $v \in W$, which contradicts with $v \in V \setminus W$. □

Lemma 7 *For any given $r \ge 1$, a subexecution that starts in round r must end by round $10r + 96 \log^2 \alpha$.*

Proof Consider any subexecution$(u, r_1, \tilde{\alpha}, \tilde{d})$ that starts in round r. By Lemma 6, subexecution$(\tilde{\alpha}, r_1, \tilde{\alpha}, \tilde{d})$ must also exist. It suffices to show that subexecution$(\tilde{\alpha}, r_1, \tilde{\alpha}, \tilde{d})$ ends by round $10r + 96 \log^2 \alpha$, since this implies that subexecution$(u, r_1, \tilde{\alpha}, \tilde{d})$ also ends by round $10r + 96 \log^2 \alpha$. During its execution of Algorithm 5, node $\tilde{\alpha}$ goes through a sequence of subexecutions. If subexecution$(\tilde{\alpha}, r_1, \tilde{\alpha}, \tilde{d})$ is the very first subexecution in this sequence, then $r = 1$ and $\tilde{d} = 2$. This implies that subexecution$(\tilde{\alpha}, r_1, \tilde{\alpha}, \tilde{d})$ must end by round $2 + 2\tilde{d} + 6\tilde{d}^2 \log^2 \tilde{\alpha} + 6\tilde{d}^3 \log^2 \tilde{\alpha} \le 10r + 96 \log^2 \alpha$. If subexecution$(\tilde{\alpha}, r_1, \tilde{\alpha}, \tilde{d})$ is not the very first subexecution in this sequence, then consider the subexecution$(\tilde{\alpha}, r_1', \tilde{\alpha}', \tilde{d}')$ on node $\tilde{\alpha}$ that is immediately before subexecution$(\tilde{\alpha}, r_1, \tilde{\alpha}, \tilde{d})$ in the sequence.

One can easily verify from the pseudo-code that $\tilde{\alpha} = \tilde{\alpha}'$ and $\tilde{d}' \geq \frac{1}{2}\tilde{d}$. Since subexecution$(\tilde{\alpha}, r_1', \tilde{\alpha}, \tilde{d}')$ takes at least $6(\tilde{d}')^2 \log^2 \tilde{\alpha} + 6(\tilde{d}')^3 \log^2 \tilde{\alpha} \geq \frac{3}{4}(\tilde{d}^2 \log^2 \tilde{\alpha} + \tilde{d}^3 \log^2 \tilde{\alpha})$ rounds, we have $r \geq \frac{3}{4}(\tilde{d}^2 \log^2 \tilde{\alpha} + \tilde{d}^3 \log^2 \tilde{\alpha})$. Next, since $\tilde{d} \geq 2$, subexecution$(\tilde{\alpha}, r_1, \tilde{\alpha}, \tilde{d})$ takes at most $2 + 2\tilde{d} + 6\tilde{d}^2 \log^2 \tilde{\alpha} + 6\tilde{d}^3 \log^2 \tilde{\alpha} \leq \frac{13}{2}(\tilde{d}^2 \log^2 \tilde{\alpha} + \tilde{d}^3 \log^2 \tilde{\alpha}) \leq 9r$ rounds. Hence subexecution$(\tilde{\alpha}, r_1, \tilde{\alpha}, \tilde{d})$ must end by round $r + 9r \leq 10r + 96 \log^2 \alpha$. □

Theorem 5 *There exists some constant c independent of d, n, and T, such that as long as $T \geq cd^2 \log^2 n$:*

- *Algorithm 5 always outputs n (and terminates) in $O(d^3 \log^2 n)$ rounds.*
- *In each round during the execution of Algorithm 5, each node sends only $O(\log n)$ bits.*

Proof All line numbers below refer to Algorithm 5. Our proof focuses on asymptotic results, without optimizing the constants. Since the nodes have ids of size $O(\log n)$, there must exist some (sufficiently large) constant c independent of n, such that $c \log^2 n \geq 400 \log^2 u$ always holds for all node id u. We will show that this value of c satisfies all the requirements in the theorem.

Time complexity. We first prove that Algorithm 5 outputs and terminates in $O(d^3 \log^2 n)$ rounds. Let r_6 be the very first round during which some node satisfies the condition at Line 17. By Lines 17 and 18, and since $T \geq cd^2 \log^2 n > d$, all nodes will terminate by round $r_6 + d$. Let $r_7 = 2300d^3 \log^2 \alpha$, hence it suffices to prove that $r_6 \leq r_7$. Prove by contradiction and assume that $r_6 > r_7$. This implies that node α (among others) does not satisfy the condition at Line 17 in the first r_7 rounds. Furthermore, by the definition of r_6, no node can possibly terminate in the first r_7 rounds.

Let $r_1 = d$. With `FloodRoot()`, every node will have seen a $\langle$`SWITCH`$, \alpha\rangle$ message by the end of round r_1. Hence starting from round $r_1 + 1$, only node α can possibly satisfy the condition at Line 5. Putting it another way, only node α can "initiate" new `SYNC` messages. Furthermore, before round $r_1 + 1$, all $\langle$`SYNC`$, \tilde{\alpha}, \tilde{d},$ `num_round`$\rangle$ message must have $\tilde{d} \leq d$. Otherwise some previous subexecution would have taken at least $6\left(\frac{\tilde{d}}{2}\right)^2 > 6\left(\frac{d}{2}\right)^2 > d = r_1$ rounds, and it would be impossible for any node to send $\langle$`SYNC`$, \tilde{\alpha}, \tilde{d}, \tilde{d} - 1\rangle$ before round $r_1 + 1$ (i.e., "initiate" the `SYNC` messages before round $r_1 + 1$). Let $r_2 = r_1 + d = 2d$. Then after round r_2, no node will ever send a $\langle$`SYNC`$, \tilde{\alpha}, \tilde{d},$ `num_round`$\rangle$ message with $\tilde{\alpha} \neq \alpha$. Hence after round r_2, no new subexecution$(u, r, \tilde{\alpha}, \tilde{d})$ with $\tilde{\alpha} \neq \alpha$ will be started. In other words, every subexecution$(u, r, \tilde{\alpha}, \tilde{d})$ where $\tilde{\alpha} \neq \alpha$ must have started before or in round r_2. By Lemma 7, all those subexecutions must end by round $r_3 = 10r_2 + 96 \log^2 \alpha = 20d + 96 \log^2 \alpha$.

Define r_4 to be the first round after round r_3 where node α sends $\langle$`SYNC`$, \alpha, d_1, d_1 - 1\rangle$ with $d_1 \geq d$. The following shows that r_4 must exist. First, let d_0 be the largest power of 2 that is no larger than d. Then for $\tilde{d}$ on node α to first reach or exceed d, it takes at most $15(2)^3 \log^2 \alpha + 15(4)^3 \log^2 \alpha + \ldots + 15(d_0)^3 \log^2 \alpha < 30d^3 \log^2 \alpha$ rounds. Next, if node α is in the middle of some subexecution in round r_3, by Lemma 7, this subexecution must end by round $10r_3 + 96 \log^2 \alpha$. Note that $10r_3 + 96 \log^2 \alpha +$

$30d^3 \log^2 \alpha + 1 < 220d^3 \log^2 \alpha$. Since $220d^3 \log^2 \alpha < r_6$, we know that r_4 must exist and that $r_4 \leq 220d^3 \log^2 \alpha$. Let subexecution$(\alpha, r_5, \alpha, d_1)$, for some r_5, be the subexecution on node α that starts in round r_4. By Lemma 7, subexecution$(\alpha, r_5, \alpha, d_1)$ must end by round $r_7 = 2300d^3 \log^2 \alpha$, since $r_7 \geq 10r_4 + 96 \log^2 \alpha$.

We will show that immediately after subexecution$(\alpha, r_5, \alpha, d_1)$ ends, the conditions at Line 17 are satisfied on node α. This would imply that $r_6 \leq r_7$, which contradicts $r_6 > r_7$ and completes the proof by contradiction. We first reason about the value of d_1. We already have $d_1 \geq d$. Recall that subexecution$(\alpha, r_5, \alpha, d_1)$ must end by round $r_7 = 2300d^3 \log^2 \alpha$. Since the execution of Line 16 in subexecution$(\alpha, r_5, \alpha, d_1)$ takes $6d_1^3 \log^2 \alpha$ round, we have $2300d^3 \log^2 \alpha \geq 6d_1^3 \log^2 \alpha$ which implies $d_1 \leq 8d$. We thus have $d \leq d_1 \leq 8d$. Let $W = \{x \mid$ subexecution(x, r_5, α, d_1) exists$\}$. Next, we show that $W = V$. Recall that every subexecution$(u, r, \tilde{\alpha}, \tilde{d})$ with $\tilde{\alpha} \neq \alpha$ must have ended by round r_3. Since $r_4 \geq r_3$, no node can be in the middle of some subexecution$(u, r, \tilde{\alpha}, \tilde{d})$ with $\tilde{\alpha} \neq \alpha$ during any round between round r_4 and round $r_4 + d_1 - 1$. Also recall that node α sends $\langle \texttt{SYNC}, \alpha, d_1, d_1 - 1\rangle$ in round r_4. Since $d_1 \geq d$, every node must receive some $\langle \texttt{SYNC}, \alpha, d_1, \texttt{num\_round}\rangle$ message in some round between round r_4 and round $r_4 + d_1 - 1$. A node u after receiving such a message will then invoke Line 14 in round r_5, which implies $W = V$.

We next show that we satisfy all the conditions needed to invoke the third clause of Lemma 4, for the invocation of `DistributeVotes()` at Line 15. One can verify from the pseudo-code that prior to this `DistributeVotes()` invocation, `ResetNeighbors()` was last invoked at the beginning of round r_5—namely, at the beginning of the very first round during the invocation of `Aggregate()` at Line 14. Let $G_2 = \sigma(r_5, r_5 + 6d_1^2 \log^2 \alpha + d_1 + 2)$. Recall that we earlier proved $d_1 \leq 8d$. Hence $T \geq cd^2 \log^2 n \geq 400d^2 \log^2 \alpha \geq 6d_1^2 \log^2 \alpha + d_1 + 3$. By the definition of the backbone diameter of T-interval dynamic networks, we immediately have $\Gamma_{G_2}(\alpha) \leq d$. Define $z = \sum_{x \in W}$(value of `votes` on node x immediately after subexecution(x, r_5, α, d_1)). We earlier showed that $W = V$ and $d_1 \geq d$. Since $W = V$, $d_1 \geq d \geq \Gamma_{G_2}(\alpha)$, $\alpha \geq n$, and since `FloodRoot` ensures that each node always send some message in each round before it terminates, we can now invoke Lemma 6, and then the third clause of Lemma 4 for the invocation of `DistributeVotes()` at Line 15. Doing so tells us that $z = \alpha^{d_1}$.

By invoking Lemma 6 and the first clause of Lemma 4, we know that the value of `votes` on node x immediately after subexecution(x, r_5, α, d_1) must be an integer. Since we further have $W = V$, $\alpha \geq n$, $z = \alpha^{d_1}$, $d_1 \geq d$, and $T \geq cd^2 \log^2 n \geq 400d^2 \log^2 \alpha \geq 3d_1^2 \log \alpha$, we invoke Lemma 6 and Eq. (11) in Lemma 3 for the invocation of `Aggregate()` at Line 16 to get `collected` $= \alpha^{d_1}$ on node α. Hence, all the conditions at Line 17 are satisfied on node α immediately after subexecution$(\alpha, r_5, \alpha, d_1)$ ends. This implies that $r_6 \leq r_7$, which contradicts $r_6 > r_7$ and completes the proof by contradiction.

Correctness. We next show that Algorithm 5 never outputs a wrong result. In order for any node to output, some node needs to satisfy the conditions at Line 17 and send the `OUTPUT` message. Recall from earlier that round r_6 was defined to be the very first round during which some node satisfies the conditions at Line 17. Let node $\tilde{\alpha}$ be any such node. Let subexecution$(\tilde{\alpha}, r, \tilde{\alpha}, \tilde{d})$ be the subexecution on node $\tilde{\alpha}$ immediately before node $\tilde{\alpha}$ satisfies the conditions. Let $W = \{x \mid$ subexecution$(x, r, \tilde{\alpha}, \tilde{d})$ exists$\}$, and we have $W \neq \emptyset$. For every $x \in W$, one can easily verify from the pseudocode that prior to the invocation of DistributeVotes($\tilde{\alpha}, \tilde{d}$) at Line 15 in subexecution($x, r, \tilde{\alpha}, \tilde{d}$), ResetNeighbors() was last invoked by node x at the beginning of round r—namely, at the beginning of the very first round during the invocation of Aggregate() at Line 14 by node x. Since node $\tilde{\alpha}$ satisfies the conditions at Line 17, on node $\tilde{\alpha}$ we must have collected $= \tilde{\alpha}^{\tilde{d}}$. By definition of r_6, no node can terminate before subexecution($\tilde{\alpha}, r, \tilde{\alpha}, \tilde{d}$) ends, and FloodRoot() further ensures that each node sends some message in each round before the node terminates. Define $z = \sum_{x \in W}$ (value of votes on node x immediately after subexecution($x, r, \tilde{\alpha}, \tilde{d}$)). Invoke Lemma 6 and the first clause in Lemma 4, and finally Eq. (10) in Lemma 3, and we will eventually have $\tilde{\alpha}^{\tilde{d}} \geq z \geq$ collected $= \tilde{\alpha}^{\tilde{d}}$. Hence we have $z = \tilde{\alpha}^{\tilde{d}}$. By Lemma 6 and the second clause of Lemma 4, we further have $W = V$ and $\tilde{d} \geq \Gamma_{G_1}(\tilde{\alpha})$, where $G_1 = \sigma(r, r + 6\tilde{d}^2 \log^2 \tilde{\alpha})$. Since $W = V$, we have $\alpha \in W$ and that subexecution($\alpha, r, \tilde{\alpha}, \tilde{d}$) exists. On the other hand, node α always satisfies the condition at Line 5. This implies that there will never be any subexecution($\alpha, r, \tilde{\alpha}, \tilde{d}$) with $\tilde{\alpha} \neq \alpha$. Hence we have $\tilde{\alpha} = \alpha$. Invoke Lemma 6 and Eq. (11) in Lemma 3 for the Aggregate() invocation at Line 14 (since $W = V$, $\tilde{d} \geq \Gamma_{G_1}(\tilde{\alpha})$, $\tilde{\alpha} = \alpha \geq n$), and we have result $= n$ on node $\tilde{\alpha}$.

Finally, we have shown above that $\tilde{\alpha} = \alpha$ and $W = V$, this means that no node can be in any subexecution after round r_6, since no node will ever send out SYNC messages after that round. Hence no other node will ever satisfy the conditions at Line 17, and there will be no other output values.

Message size. We finally prove that in each round during the execution of Algorithm 5, each node u sends only $O(\log n)$ bits. Recall from earlier that r_6 is the very first round during which some node satisfies the condition at Line 17. We have shown in the above that no node can be in any subexecution after round r_6. Hence after round r_6, trivially, each node u sends only $O(\log n)$ bits per round.

Before or in round r_6, we focus on the number of bits sent in each round by node u at Lines 14, 15, and 16—one can easily but tediously verify that the number of bits sent by node u at all other lines of the algorithm is always $O(\log n)$. Consider any subexecution($u, r, \tilde{\alpha}, \tilde{d}$) during which node u invokes Line 14, Line 15, or Line 16. Let $W = \{x \mid \text{subexecution}(x, r, \tilde{\alpha}, \tilde{d}) \text{ exists}\}$. Then we have $u \in W$ and $W \neq \emptyset$. By Lemmas 6, 3, and 4, we know that the number of bits sent in each round by node u at Lines 14, 15, and 16 must all be $O(\log n)$. □

4 Our Algorithms for Other Problems

This section discusses how we use our novel approach/framework to solve a range of other problems beyond Count, when $T \geq cd^2 \log^2 n$. For solving Max/LeaderElect/Consensus/ConfirmedFlood, we only need to replace Line 14 in Algorithm 5 in the following way. Instead of invoking Aggregate(), Line 14 will now simply flood, for $2\tilde{d}$ rounds, the maximum input value seen (for Max and Consensus) or the maximum node id seen (for LeaderElect) or the input of the distinguished node (for ConfirmedFlood). When the condition at Line 17 is satisfied, $\tilde{d}$ must have been large enough, and hence every node must have previously in Line 14 "heard from" all nodes. This then enables the algorithm to output a correct result in $O(d^3 \log^2 n)$ rounds.

For solving SUM and MEDIAN, we first invoke the COUNT algorithm to obtain n. Note that when the algorithm terminates, all nodes must already see the largest id α among all the n nodes. Hence in all future invocations of the algorithm, there will only be a single "instance"—namely, the "instance" whose root is α. Next, we invoke the algorithm again to get the maximum value z among the n input values. Finally, for SUM, we invoke the algorithm a third time while changing Line 14 from "`result` $\leftarrow$ `Aggregate`$(\tilde{\alpha}, \tilde{d}, 1, \tilde{\alpha}, \textbf{false})$" to "`result` $\leftarrow$ `Aggregate`$(\tilde{\alpha}, \tilde{d}, x, nz, \textbf{false})$", with x being the local input on the node. Doing so solves SUM in $O(d^3 \log^2 n)$ rounds. For MEDIAN, after getting n and z, node α does a binary search in the range of $[0, z]$. In each step of the search, node α indicates the current range of interest, and uses the algorithm to count the number of inputs falling within that range. This solves MEDIAN in $O(d^3 \log^3 n)$ rounds.

Acknowledgements We thank Ruomu Hou, the anonymous SPAA 2020 reviewers, and the anonymous Algorithmica reviewers, for their helpful feedback on this paper.

Funding This research is partly supported by the Ministry of Education, Singapore, under its MOE Academic Research Fund Tier 2 (research Grant No.: MOE2017-T2-2-031). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore. Irvan Jahja has no relevant financial or non-financial interests to disclose. Haifeng Yu is an Associate Professor in School of Computing, National University of Singapore (NUS).

References

1. Jahja, I., Yu, H.: Sublinear algorithms in T-interval dynamic networks. In: SPAA (2020)
2. Abshoff, S., Benter, M., Cord-Landwehr, A., Malatyali, M., Meyer auf der Heide, F.: Token dissemination in geometric dynamic networks. In: ALGOSENSORS (2013)
3. Ahmadi, M., Kuhn, F.: Multi-message broadcast in dynamic radio networks. In: ALGOSENSORS (2017)
4. Brandes, P., Meyer auf der Heide, F.: Distributed computing in fault-prone dynamic networks. In: International Workshop on Theoretical Aspects of Dynamic Distributed Systems (2012)
5. Das Sarma, A., Molla, A., Pandurangan, G.: Fast distributed computation in dynamic networks via random walks. In: DISC (2012)
6. Haeupler, B., Karger, D.: Faster information dissemination in dynamic networks via network coding. In: PODC (2011)
7. Kuhn, F., Lynch, N., Oshman, R.: Distributed computation in dynamic networks. In: STOC (2010)
8. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania (1987)
9. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
10. Yu, H., Zhao, Y., Jahja, I.: The cost of unknown diameter in dynamic networks. J. ACM **65**(5), 31–13134 (2018)
11. Chakraborty, M., Milani, A., Mosteiro, M.: A faster exact-counting protocol for anonymous dynamic networks. Algorithmica **80**(11), 3023–3049 (2018)
12. Kowalski, D., Mosteiro, M.: Polynomial counting in anonymous dynamic networks with applications to anonymous dynamic algebraic computations. In: ICALP (2018)
13. Luna, G., Baldoni, R., Bonomi, S., Chatzigiannakis, I.: Conscious and unconscious counting on anonymous dynamic networks. In: International Conference on Distributed Computing and Networking (2014)

14. Luna, G., Baldoni, R., Bonomi, S., Chatzigiannakis, I.: Counting in anonymous dynamic networks under worst-case adversary. In: IEEE International Conference on Distributed Computing Systems (2014)
15. Luna, G., Bonomi, S., Chatzigiannakis, I., Baldoni, R.: Counting in anonymous dynamic networks: an experimental perspective. In: ALGOSENSORS (2013)
16. Michail, O., Chatzigiannakis, I., Spirakis, P.: Naming and counting in anonymous unknown dynamic networks. In: Proceedings of International Symposium on Stabilization, Safety, and Security of Distributed Systems (2013)
17. Chen, B., Yu, H., Zhao, Y., Gibbons, P.B.: The cost of fault tolerance in multi-party communication complexity. J. ACM **61**(3), 19–11964 (2014)
18. Kuhn, F., Oshman, R.: The complexity of data aggregation in directed networks. In: DISC (2011)
19. Kuhn, F., Moses, Y., Oshman, R.: Coordinated consensus in dynamic networks. In: PODC (2011)
20. Charron-Bost, B., Fugger, M., Nowak, T.: Approximate consensus in highly dynamic networks: the role of averaging algorithms. In: ICALP (2015)
21. Coulouma, E., Godard, E.: A characterization of dynamic networks where consensus is solvable. In: SIROCCO (2013)
22. Schmid, U., Weiss, B., Keidar, I.: Impossibility results and lower bounds for consensus under link failures. SIAM J. Comput. **38**(5), 1912–1951 (2009)
23. Augustine, J., Pandurangan, G., Robinson, P.: Fast byzantine agreement in dynamic networks. In: PODC (2013)
24. Augustine, J., Pandurangan, G., Robinson, P.: Fast byzantine leader election in dynamic networks. In: DISC (2015)
25. Ingram, R., Shields, P., Walter, J.: An asynchronous leader election algorithm for dynamic networks. In: IPDPS (2009)
26. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
27. Chlebus, B., Kowalski, D., Olkowski, J., Olkowski, J.: Disconnected agreement in networks prone to link failures. In: International Symposium on Stabilizing, Safety, and Security of Distributed Systems (2023)
28. Hou, R., Jahja, I., Sun, Y., Wu, J., Yu, H.: Achieving sublinear complexity under constant T in T-interval dynamic networks. In: SPAA (2022)
29. Kuhn, F., Oshman, R.: Dynamic networks: models and algorithms. SIGACT News **42**(1), 82–96 (2011)
30. Almeida, P., Baquero, C., Farach-Colton, M., Jesus, P., Mosteiro, M.A.: Fault-tolerant aggregation: flow-updating meets mass-distribution. Distributed Comput. **30**(4), 281–291 (2017)
31. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: FOCS (2003)
32. Terpstra, W.W., Leng, C., Buchmann, A.P.: Brief announcement: practical summation via gossip. In: PODC (2007)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.