Efficient Distributed Random Walks with Applications

Atish Das Sarma
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
atish@cc.gatech.edu

Danupon Nanongkai
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
danupon@cc.gatech.edu

Gopal Pandurangan*
Division of Mathematical Sciences, Nanyang Technological University, Singapore 637371
gopalpandurangan@gmail.com

Prasad Tetali†
School of Mathematics and School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA
tetali@math.gatech.edu

ABSTRACT

We focus on the problem of performing random walks efficiently in a distributed network. Given bandwidth constraints, the goal is to minimize the number of rounds required to obtain a random walk sample. We first present a fast sublinear time distributed algorithm for performing random walks whose time complexity is sublinear in the length of the walk. Our algorithm performs a random walk of length ℓ in $\tilde{O}(\sqrt{\ell D})$ rounds (with high probability) on an undirected network, where D is the diameter of the network. This improves over the previous best algorithm that ran in $\tilde{O}(\ell^{2/3}D^{1/3})$ rounds (Das Sarma et al., PODC 2009). We further extend our algorithms to efficiently perform k independent random walks in $\tilde{O}(\sqrt{k\ell D} + k)$ rounds. We then show that there is a fundamental difficulty in improving the dependence on ℓ any further by proving a lower bound of $\Omega(\sqrt{\frac{\ell}{\log \ell}} + D)$ under a general model of distributed random walk algorithms. Our random walk algorithms are useful in speeding up distributed algorithms for a variety of applications that use random walks as a subroutine. We present two main applications. First, we give a fast distributed algorithm for computing a random spanning tree (RST) in an arbitrary (undirected) network which runs in $\tilde{O}(\sqrt{m}D)$ rounds (with high probability; here m is the number of edges). Our second application is a fast decentralized algorithm for estimating mixing time and related parameters of the underlying network. Our algorithm is fully decentralized and can serve as a building block in the design of topologically-aware networks.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]:
Nonnumerical Algorithms and Problems—*computations on discrete structures*;
G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*;
G.2.2 [**Discrete Mathematics**]: Graph Theory—*network problems*

General Terms

Algorithms, Theory

Keywords

Random walks, Random sampling, Decentralized computation, Distributed algorithms, Random Spanning Tree, Mixing Time.

1. INTRODUCTION

Random walks play a central role in computer science, spanning a wide range of areas in both theory and practice. The focus of this paper is random walks in networks, in particular, decentralized algorithms for performing random walks in arbitrary networks. Random walks are used as an integral subroutine in a wide variety of network applications ranging from token management and load balancing to search, routing, information propagation and gathering, network topology construction and building random spanning trees (e.g., see [11] and the references therein). Random walks are also very useful in providing uniform and efficient solutions to distributed control of dynamic networks [8, 33]. Random walks are local and lightweight and require little index or state maintenance which make them especially attractive to self-organizing dynamic networks such as Internet overlay and ad hoc wireless networks.

*Also affiliated with Department of Computer Science, Brown University, Providence, RI 02912, USA. Supported in part by NSF grant CCF-0830476
†Supported in part by NSF DMS 0701023 and NSF CCR 0910584

PODC'10, July 25–28, 2010, Zurich, Switzerland.

A key purpose of random walks in many of these network applications is to perform node sampling. While the sampling requirements in different applications vary, whenever a true sample is required from a random walk of certain steps, typically all applications perform the walk naively — by simply passing a token from one node to its neighbor: thus to perform a random walk of length ℓ takes time linear in ℓ.

In this paper, we present a sublinear time (sublinear in ℓ) distributed random walk sampling algorithm that is significantly faster than the previous best result. Our algorithm runs in time $\tilde{O}(\sqrt{\ell D})$ rounds. We then present an almost matching lower bound that applies to a general class of distributed algorithms (our algorithm also falls in this class). Finally, we present two key applications of our algorithm. The first is a fast distributed algorithm for computing a random spanning tree, a fundamental spanning tree problem that has been studied widely in the classical setting (see e.g., [21] and references therein). To the best of our knowledge, our algorithm gives the fastest known running time in an arbitrary network. The second is to devising efficient decentralized algorithms for computing key global metrics of the underlying network — mixing time, spectral gap, and conductance. Such algorithms can be useful building blocks in the design of *topologically (self-)aware* networks, i.e., networks that can monitor and regulate themselves in a decentralized fashion. For example, efficiently computing the mixing time or the spectral gap, allows the network to monitor connectivity and expansion properties of the network.

1.1 Distributed Computing Model

Consider an undirected, unweighted, connected n–node graph $G = (V, E)$. Suppose that every node (vertex) hosts a processor with unbounded computational power, but with limited initial knowledge. Specifically, assume that each node is associated with a distinct identity number from the set $\{1, 2, ..., n\}$. At the beginning of the computation, each node v accepts as input its own identity number and the identity numbers of its neighbors in G. The node may also accept some additional inputs as specified by the problem at hand. The nodes are allowed to communicate through the edges of the graph G. The communication is synchronous, and occurs in discrete pulses, called *rounds*. In particular, all the nodes wake up simultaneously at the beginning of round 1, and from this point on the nodes always know the number of the current round. In each round each node v is allowed to send an arbitrary message of size $O(\log n)$ through each edge $e = (v, u)$ that is adjacent to v, and the message will arrive to u at the end of the current round. This is a standard model of distributed computation known as the *CONGEST model* [29] and has been attracting a lot of research attention during last two decades (e.g., see [29] and the references therein).

There are several measures of efficiency of distributed algorithms, but we will concentrate on one of them, specifically, *the running time*, that is, the number of rounds of distributed communication. (Note that the computation that is performed by the nodes locally is "free", i.e., it does not affect the number of rounds.) Many fundamental network problems such as minimum spanning tree, shortest paths, etc. have been addressed in this model (e.g., see [26, 29, 28]). In particular, there has been much research into designing very fast distributed approximation algorithms (that are even faster at the cost of producing sub-optimal solutions) for many of these problems (see e.g., [14, 13, 24, 23]). Such algorithms can be useful for large-scale resource-constrained and dynamic networks where running time is crucial.

1.2 Problem Statement, Motivation, and Related Work

The basic problem we address is the following. We are given an arbitrary undirected, unweighted, and connected n–node network $G = (V, E)$ and a (source) node $s \in V$. The goal is to devise a distributed algorithm such that, in the end, s outputs the ID of a node v which is randomly picked according to the probability that it is the destination of a random walk of length ℓ starting at s. Throughout this paper, we assume the standard (simple) random walk: in each step, an edge is taken from the current node x with probability proportional to $1/d(x)$ where $d(x)$ is the degree of x. Our goal is to output a true random sample from the ℓ-walk distribution starting from s.

For clarity, observe that the following naive algorithm solves the above problem in $O(\ell)$ rounds: The walk of length ℓ is performed by sending a token for ℓ steps, picking a random neighbor with each step. Then, the destination node v of this walk sends its ID back (along the same path) to the source for output. Our goal is to perform such sampling with significantly less number of rounds, i.e., in time that is sublinear in ℓ. On the other hand, we note that it can take too much time (as much as $\Theta(|E| + D)$ time) in the CONGEST model to collect all the topological information at the source node (and then computing the walk locally).

This problem was proposed in [11] under the name *Computing One Random Walk where Source Outputs Destination (1-RW-SoD)* (for short, this problem will be simply called *Single Random Walk* in this paper), wherein the first sublinear time distributed algorithm was provided, requiring $\tilde{O}(\ell^{2/3}D^{1/3})$ rounds ($\tilde{O}$ hides polylog(n) factors); this improves over the naive $O(\ell)$ algorithm when the walk is long compared to the diameter (i.e., $\ell = \Omega(D \operatorname{polylog} n)$ where D is the diameter of the network). This was the first result to break past the inherent sequential nature of random walks and beat the naive ℓ round approach, despite the fact that random walks have been used in distributed networks for long and in a wide variety of applications.

There are two key motivations for obtaining sublinear time bounds. The first is that in many algorithmic applications, walks of length significantly greater than the network diameter are needed. For example, this is necessary in both the applications presented later in the paper, namely distributed computation of a random spanning tree (RST) and computation of mixing time. In the RST algorithm, we need to perform a random walk of expected length $O(mD)$ (where m is the number of edges in the network). In decentralized computation of mixing time, we need to perform walks of length at least equal to the mixing time which can be significantly larger than the diameter (e.g., in a random geometric graph model [27], a popular model for ad hoc networks, the mixing time can be larger than the diameter by a factor of $\Omega(\sqrt{n})$.) More generally, many real-world communication networks (e.g., ad hoc networks and peer-to-peer networks) have relatively small diameter, and random walks of length at least the diameter are usually performed for many sampling applications, i.e., $\ell >> D$. It should be noted that if the network is rapidly mixing/expanding which is sometimes the case in

practice, then sampling from walks of length $\ell >> D$ is close to sampling from the steady state (degree) distribution; this can be done in $O(D)$ rounds (note however, that this gives only an approximately close sample, not the exact sample for that length). However, such an approach fails when ℓ is smaller than the mixing time.

The second motivation is understanding the time complexity of distributed random walks. Random walk is essentially a global problem which requires the algorithm to "traverse" the entire network. Classical "global" problems include the minimum spanning tree, shortest path etc. Network diameter is an inherent lower bound for such problems. Problems of this type raise the basic question whether n (or ℓ as the case here) time is essential or is the network diameter D, the inherent parameter. As pointed out in the seminal work of [17], in the latter case, it would be desirable to design algorithms that have a better complexity for graphs with low diameter.

The high-level idea used in the $\tilde{O}(\ell^{2/3}D^{1/3})$-round algorithm in [11] is to "prepare" a few short walks in the beginning (executed in parallel) and then carefully stitch these walks together later as necessary. The same general approach was introduced in [10] to find random walks in data streams with the main motivation of finding PageRank. However, the two models have very different constraints and motivations and hence the subsequent techniques used in [11] and [10] are very different.

Recently, Sami and Twigg [31] consider lower bounds on the communication complexity of computing stationary distribution of random walks in a network. Although, their problem is related to our problem, the lower bounds obtained do not imply anything in our setting. Other recent works involving multiple random walks in different settings include Alon et. al. [3], Elsässer et. al. [16], and Cooper et al. [9].

1.3 Our Results

- **A Fast Distributed Random Walk Algorithm:** We present a sublinear, almost time-optimal, distributed algorithm for the single random walk problem in arbitrary networks that runs in time $\tilde{O}(\sqrt{\ell D})$, where ℓ is the length of the walk (cf. Section 2). This is a significant improvement over the naive ℓ-round algorithm for $\ell = \Omega(D)$ as well as over the previous best running time of $\tilde{O}(\ell^{2/3}D^{1/3})$ [11]. The dependence on ℓ is reduced from $\ell^{2/3}$ to $\ell^{1/2}$.

  Our algorithm in this paper uses an approach similar to that of [11] but exploits certain key properties of random walks to design an even faster sublinear time algorithm. Our algorithm is randomized (Las Vegas type, i.e., it always outputs the correct result, but the running time claimed is with high probability) and is conceptually simpler compared to the $\tilde{O}(\ell^{2/3}D^{1/3})$-round algorithm. While the previous (slower) algorithm [11] applies to the more general Metropolis-Hastings walk, in this work we focus primarily on the simple random walk for the sake of obtaining the best possible bounds in this commonly used setting.

  One of the key ingredients in the improved algorithm is proving a bound on the number of times any node is visited in an ℓ-length walk, for any length $\ell = O(m^2)$. We show that w.h.p. any node x is visited at most $\tilde{O}(d(x)\sqrt{\ell})$ times, in an ℓ-length walk from any starting node ($d(x)$ is the degree of x). We then show that if only certain ℓ/λ special points of the walk (called as *connector points*) are observed, then any node is observed only $\tilde{O}(d(x)\sqrt{\ell}/\lambda)$ times. The algorithm starts with all nodes performing short walks (of length uniformly random in the range λ to 2λ for appropriately chosen λ) efficiently simultaneously; here the randomly chosen lengths play a crucial role in arguing about a suitable spread of the connector points. Subsequently, the algorithm begins at the source and carefully stitches these walks together till ℓ steps are completed.

  We also extend to give algorithms for computing k random walks (from any k sources —not necessarily distinct) in $\tilde{O}\left(\min(\sqrt{k\ell D}+k, k+\ell)\right)$ rounds. Computing k random walks is useful in many applications such as the one we present below on decentralized computation of mixing time and related parameters. While the main requirement of our algorithms is to just obtain the random walk samples (i.e. the end point of the ℓ step walk), our algorithms can regenerate the entire walks such that each node knows its position(s) among the ℓ steps. Our algorithm can be extended to do this in the same number of rounds.

- **A Lower Bound:** We establish an almost matching lower bound on the running time of distributed random walk that applies to a general class of distributed random walk algorithms. We show that any algorithm belonging to the class needs at least $\Omega(\sqrt{\frac{\ell}{\log \ell}}+D)$ rounds to perform a random walk of length ℓ; notice that this lower bound is nontrivial even in graphs of small ($D = O(\log n)$) diameter (cf. Section 3). Broadly speaking, we consider a class of token forwarding-type algorithms where nodes can only store and (selectively) forward tokens (here tokens are $O(\log n)$-sized messages consisting of two node ids identifying the beginning and end of a segment — we make this more precise in Section 3). Selective forwarding (more general than just store and forwarding) means that nodes can omit to forward certain segments (to reduce number of messages), but they cannot alter tokens in any way (e.g., resort to data compression techniques). This class includes many natural algorithms, including the algorithm in this paper.

  Our technique involves showing the same non-trivial lower bound for a problem that we call *path verification*. This simpler problem appears quite basic and can have other applications. Informally, given a graph G and a sequence of ℓ vertices in the graph, the problem is for some (source) node in the graph to verify that the sequence forms a path. One main idea in this proof is to show that independent nodes may be able to verify short *local* paths; however, to be able to *merge* these together and verify an ℓ-length path would require exchanging several messages. The trade-off is between the lengths of the local paths that are verified and the number of such local paths that need to be combined. Locally verified paths can be exchanged in one round, and messages can be exchanged at all nodes. Despite this, we show that the bandwidth restriction necessitates a large number of rounds even if the diameter is small. We then show a reduction to the random walk problem, where we require that each node

in the walk should know its (correct) position(s) in the walk.

Similar non-trivial matching lower bounds on running time are known only for a few important problems in distributed computing, notably the minimum spanning tree problem (e.g., see [30, 15]). Peleg and Rabinovich [30] showed that $\tilde{\Omega}(\sqrt{n})$ time is required for constructing an MST even on graphs of small diameter (for any $D = \Omega(\log n)$) and [25] showed an essentially matching upper bound.

- **Applications:** Our faster distributed random walk algorithm can be used in speeding up distributed applications where random walks arise as a subroutine. Such applications include distributed construction of expander graphs, checking whether a graph is an expander, construction of random spanning trees, and random-walk based search (we refer to [11] for details). Here, we present two key applications:

  (1) *A Fast Distributed Algorithm for Random Spanning Trees (RST):* We give a $\tilde{O}(\sqrt{m}D)$ time distributed algorithm (cf. Section 4.1) for uniformly sampling a random spanning tree in an arbitrary undirected (unweighted) graph (i.e., each spanning tree in the underlying network has the same probability of being selected). (m denotes the number of edges in the graph.) Spanning trees are fundamental network primitives and distributed algorithms for various types of spanning trees such as minimum spanning tree (MST), breadth-first spanning tree (BFS), shortest path tree, shallow-light trees etc., have been studied extensively in the literature [29]. However, not much is known about the distributed complexity of the random spanning tree problem. The centralized case has been studied for many decades, see e.g., the recent work of [21] and the references therein; also see the recent work of Goyal et al. [19] which gives nice applications of RST to fault-tolerant routing and constructing expanders. In the distributed context, the work of Bar-Ilan and Zernik [5] give a distributed RST algorithm for two special cases, namely that of a complete graph (running in constant time) and a synchronous ring (running in $O(n)$ time). The work of [4] give a self-stablizing distributed algorithm for constructing a RST in a wireless ad hoc network and mentions that RST is more resilient to transient failures that occur in mobile ad hoc networks.

  Our algorithm works by giving an efficient distributed implementation of the well-known Aldous-Broder random walk algorithm [1, 7] for constructing a RST.

  (2) *Decentralized Computation of Mixing Time.* We present a fast decentralized algorithm for estimating mixing time, conductance and spectral gap of the network (cf. 4.2). In particular, we show that given a starting point x, the mixing time with respect to x, called τ^x_{mix}, can be estimated in $\tilde{O}(n^{1/2} + n^{1/4}\sqrt{D\tau^x_{mix}})$ rounds. This gives an alternative algorithm to the only previously known approach by Kempe and McSherry [22] that can be used to estimate τ^x_{mix} in $\tilde{O}(\tau^x_{mix})$ rounds.[1] To compare, we note that when $\tau^x_{mix} = \omega(n^{1/2})$ the present algorithm is faster (assuming D is not too large).

  The work of [18] discusses spectral algorithms for enhancing the topology awareness, e.g., by identifying and assigning weights to critical links. However, the algorithms are centralized, and it is mentioned that obtaining efficient decentralized algorithms is a major open problem. Our algorithms are fully decentralized and based on performing random walks, and so more amenable to dynamic and self-organizing networks.

[1]Note that [22] in fact do more and give a decentralized algorithm for computing the top k eigenvectors of a weighted adjacency matrix that runs in $O(\tau_{mix} \log^2 n)$ rounds if two adjacent nodes are allowed to exchange $O(k^3)$ messages per round, where τ_{mix} is the mixing time and n is the size of the network.

2. A SUBLINEAR TIME DISTRIBUTED RANDOM WALK ALGORITHM

2.1 Description of the Algorithm

We first describe the $\tilde{O}(\ell^{2/3}D^{1/3})$-round algorithm in [11] and then highlight the changes in our current algorithm. The current algorithm is randomized and uses several new ideas that are crucial in obtaining the new bound.

The high-level idea is to perform "many" short random walks in parallel and later stitch them together as needed. In the first phase of the algorithm Single-Random-Walk (we refer to the full version [12] for pseudocodes of all algorithms and subroutines), each node performs η independent random walks of length λ. (Only the destination of each of these walks is aware of its source, but the sources do not know destinations right away. The sources will get to know destinations later on when it is needed.) It is shown that this takes $\tilde{O}(\eta\lambda)$ rounds with high probability. Subsequently, the source node that requires a walk of length ℓ extends a walk of length λ by "stitching" walks. If the end point of the first λ length walk is u, one of u's λ length walks is used to extend. When at u, one of its λ-length walk destinations are sampled uniformly (to preserve randomness) using Sample-Destination in $O(D)$ rounds (including the time to deliver such sampled destination to u). (We call such u and other nodes at the stitching points as *connectors* — cf. Algorithm 1.) Each stitch takes $O(D)$ rounds (via the BFS tree). This process is extended as long as unused λ-length walks are available from visited nodes. If the walk reaches a node v where all η walks have been used up (which is a key difficulty), then Get-More-Walks is invoked. Get-More-Walks performs η more walks of length λ from v, and this can be done in $\tilde{O}(\lambda)$ rounds. The number of times Get-More-Walks is invoked can be bounded by $\frac{\ell}{\eta\lambda}$ in the worst case by an amortization argument. The overall bound on the algorithm is $O(\eta\lambda + \ell D/\lambda + \frac{\ell}{\eta})$. The bound of $\tilde{O}(\ell^{2/3}D^{1/3})$ follows from appropriate choice of parameters η and λ.

The current algorithm uses two crucial ideas to improve the running time. The first idea is to bound the number of times any node is visited in a random walk of length ℓ (which in turn bounds the number of times Get-More-Walks is invoked). Instead of the worst case analysis in [11], the new bound is obtained by bounding the number of times any node is visited (with high probability) in a random walk of length ℓ on an undirected unweighted graph. The number of visits to a node beyond the mixing time can be bounded using its stationary probability distribution. However, we

need a bound on the visits to a node for any ℓ-length walk starting from the first step. We show a somewhat surprising bound that applies to an ℓ-length (for $\ell = O(m^2)$) random walk on any arbitrary (undirected) graph: *no node x is visited more than $\tilde{O}(d(x)\sqrt{\ell})$ times*, in an ℓ-length walk from any starting node ($d(x)$ is the degree of x) (cf. Lemma 2.6). Note that this bound does not depend on any other parameter of the graph, just on the (local) degree of the node and the length of the walk. This bound is tight in general (e.g., consider a line and a walk of length n).

The above bound is not enough to get the desired running time, as it does not say anything about the distribution of connectors when we chop the length ℓ walk into ℓ/λ pieces. We have to bound the number of visits to a node as a connector in order to bound the number of times Get-More-Walks is invoked. To overcome this we use a second idea: Instead of nodes performing walks of length λ, each such walk i is of length $\lambda + r_i$ where r_i is a random number in the range $[0, \lambda - 1]$. Notice that the random numbers are independent for each walk. We show the following "uniformity lemma": if the short walks are now of a random length in the range of $[\lambda, 2\lambda - 1]$, then if a node u is visited at most N_u times in an ℓ step walk, then the node is visited at most $\tilde{O}(N_u/\lambda)$ times as an endpoint of a short walk (cf. Lemma 2.7). This modification to Single-Random-Walk allows us to bound the number of visits to each node (cf. Lemma 2.7).

The change of the short walk length above leads to two modifications in Phase 1 of Single-Random-Walk and Get-More-Walks. In Phase 1, generating η walks of different lengths from each node is straightforward: Each node simply sends η tokens containing the source ID and the desired length. The nodes keep forwarding these tokens with decreased desired walk length until the desired length becomes zero. The modification of Get-More-Walks is trickier. To avoid congestion, we use the idea of *reservoir sampling* [32]. In particular, we add the following process at the end of the Get-More-Walks algorithm in [11]:

for $i = 0$ to $\lambda - 1$ **do**
  For each message, independently with probability $\frac{1}{\lambda - i}$, stop sending the message further and save the ID of the source node (in this event, the node with the message is the destination). For messages M that are not stopped, each node picks a neighbor correspondingly and sends the messages forward as before.
end for

The reason it needs to be done this way is that if we first sampled the walk length r, independently for each walk, in the range $[0, \lambda - 1]$ and then extended each walk accordingly, the algorithm would need to pass r independently for each walk. This will cause congestion along the edges; no congestion occurs in the mentioned algorithm as only the *count* of the number of walks along an edge are passed to the node across the edge. Therefore, we need to decide when to stop on the fly using reservoir sampling.

We also have to make another modification in Phase 1 due to the new bound on the number of visits. Recall that, in this phase, each node prepares η walks of length λ. However, since the new bound of visits of each node x is proportional to its degree $d(x)$ (see Lemma 2.6), we make each node prepare $\eta d(x)$ walks instead. We show that Phase 1 uses $\tilde{O}(\eta\lambda)$ rounds, instead of $\tilde{O}(\frac{\lambda\eta}{\delta})$ rounds where δ is the minimum degree in the graph (cf. Lemma 2.3).

To summarize, the main algorithm for performing a single random walk is Single-Random-Walk. This algorithm, in turn, uses Get-More-Walks and Sample-Destination. The key modification is that, instead of creating short walks of length λ each, we create short walks where each walk has length in range $[\lambda, 2\lambda - 1]$. To do this, we modify the Phase 1 of Single-Random-Walk and Get-More-Walks.

We now state four lemmas which are similar to the Lemma 2.2-2.6 in [11]. However, since the algorithm here is a modification of that in [11], we include the full proofs in the full version [12].

Lemma 2.1. *Phase 1 finishes in $O(\lambda\eta \log n)$ rounds with high probability.*

Lemma 2.2. *For any v,* Get-More-Walks(v, η, λ) *always finishes within $O(\lambda)$ rounds.*

Lemma 2.3. Sample-Destination *always finishes within $O(D)$ rounds.*

Lemma 2.4. *Algorithm* Sample-Destination*(v) returns a destination from a random walk whose length is uniform in the range $[\lambda, 2\lambda - 1]$.*

2.2 Analysis

The following theorem states the main result of this Section. It states that the algorithm Single-Random-Walk correctly samples a node after a random walk of ℓ steps and the algorithm takes, with high probability, $\tilde{O}\left(\sqrt{\ell D}\right)$ rounds where D is the diameter of the graph. Throughout this section, we assume that ℓ is $O(m^2)$, where m is the number of edges in the network. If ℓ is $\Omega(m^2)$, the required bound is easily achieved by aggregating the graph topology (via upcast) onto one node in $O(m + D)$ rounds (e.g., see [29]). The difficulty lies in proving for $\ell = O(m^2)$.

Theorem 2.5. *For any ℓ, Algorithm* Single-Random-Walk *solves 1-RW-DoS (the Single Random Walk Problem) and, with probability at least $1 - \frac{2}{n}$, finishes in $\tilde{O}\left(\sqrt{\ell D}\right)$ rounds.*

We prove the above theorem using the following lemmas. As mentioned earlier, to bound the number of times Get-More-Walks is invoked, we need a technical result on random walks that bounds the number of times a node will be visited in a ℓ-length random walk. Consider a simple random walk on a connected undirected graph on n vertices. Let $d(x)$ denote the degree of x, and let m denote the number of edges. Let $N_t^x(y)$ denote the number of visits to vertex y by time t, given the walk started at vertex x. Now, consider k walks, each of length ℓ, starting from (not necessary distinct) nodes $x_1, x_2, \ldots, x_k$. We show a key technical lemma (proof in the full version [12]) that applies to a random walk on any graph: With high probability, no vertex y is visited more than $24d(x)\sqrt{k\ell + 1}\log n + k$ times.

Lemma 2.6. *For any nodes $x_1, x_2, \ldots, x_k$, and $\ell = O(m^2)$,*

$$\Pr\left(\exists y \text{ s.t. } \sum_{i=1}^{k} N_\ell^{x_i}(y) \geq 24d(x)\sqrt{k\ell + 1}\log n + k\right) \leq 1/n\,.$$

This lemma says that the number of visits to each node can be bounded. However, for each node, we are only interested in the case where it is used as a connector. The lemma below shows that the number of visits as a connector can be bounded as well; i.e., if any node v_i appears t times in the walk, then it is likely to appear roughly t/λ times as connectors.

LEMMA 2.7. *For any vertex v, if v appears in the walk at most t times then it appears as a connector node at most $t(\log n)^2/\lambda$ times with probability at least $1 - 1/n^2$.*

Intuitively, this argument is simple, since the connectors are spread out in steps of length approximately λ. However, there might be some *periodicity* that results in the same node being visited multiple times but *exactly* at λ-intervals. This is where we crucially use the fact that the algorithm uses walks of length $\lambda + r$ where r is chosen uniformly at random from $[0, \lambda-1]$. The proof then goes via constructing another process equivalent to partitioning the ℓ steps in to intervals of λ and then sampling points from each interval. We analyze this by carefully constructing a different process that stochastically dominates the process of a node occurring as a connector at various steps in the ℓ-length walk and then use a Chernoff bound argument. The detailed proof is presented in the full version [12].

Now we are ready to prove Theorem 2.5.

PROOF OF THEOREM 2.5. First, we claim, using Lemma 2.6 and 2.7, that each node is used as a connector node at most $\frac{24d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ times with probability at least $1 - 2/n$. To see this, observe that the claim holds if each node x is visited at most $t(x) = 24d(x)\sqrt{\ell+1}\log n$ times and consequently appears as a connector node at most $t(x)(\log n)^2/\lambda$ times. By Lemma 2.6, the first condition holds with probability at least $1 - 1/n$. By Lemma 2.7 and the union bound over all nodes, the second condition holds with probability at least $1 - 1/n$, provided that the first condition holds. Therefore, both conditions hold together with probability at least $1 - 2/n$ as claimed.

Now, we choose $\eta = 1$ and $\lambda = 24\sqrt{\ell D}(\log n)^3$. By Lemma 2.1, Phase 1 finishes in $\tilde{O}(\lambda\eta) = \tilde{O}(\sqrt{\ell D})$ rounds with high probability. For Phase 2, SAMPLE-DESTINATION is invoked $O(\frac{\ell}{\lambda})$ times (only when we stitch the walks) and therefore, by Lemma 2.3, contributes $O(\frac{\ell D}{\lambda}) = \tilde{O}(\sqrt{\ell D})$ rounds. Finally, we claim that GET-MORE-WALKS is never invoked, with probability at least $1 - 2/n$. To see this, recall our claim above that each node is used as a connector node at most $\frac{24d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ times. Moreover, observe that we have prepared this many walks in Phase 1; i.e., after Phase 1, each node has $\eta\lambda d(x) = \frac{24d(x)\sqrt{\ell}(\log n)^3}{\lambda}$ short walks. The claim follows.

Therefore, with probability at least $1 - 2/n$, the rounds are $\tilde{O}(\sqrt{\ell D})$ as claimed. □

Regenerating the entire random walk: It is important to note that our algorithm can be extended to regenerate the entire walk. As described above, the source node obtains the sample after a random walk of length ℓ. In certain applications, it may be desired that the entire random walk be obtained, i.e., every node in the ℓ length walk knows its position(s) in the walk. This can be done by first informing all intermediate connecting nodes of their position (since there are only $O(\sqrt{\ell})$ such nodes). Then, these nodes can regenerate their $O(\sqrt{\ell})$ length short walks by simply sending a message through each of the corresponding short walks. This can be completed in $\tilde{O}(\sqrt{\ell D})$ rounds with high probability. This is because, with high probability, GET-MORE-WALK will not be invoked and hence all the short walks are generated in Phase 1. Sending a message through each of these short walks (in fact, sending a message through *every* short walk generated in Phase 1) takes time at most the time taken in Phase 1, i.e., $\tilde{O}(\sqrt{\ell D})$ rounds.

2.3 Extension to Computing k Random Walks

We now consider the scenario when we want to compute k walks of length ℓ from different (not necessary distinct) sources $s_1, s_2, \ldots, s_k$. We show that SINGLE-RANDOM-WALK can be extended to solve this problem. Consider the following algorithm.

MANY-RANDOM-WALKS∴.

Let $\lambda = (24\sqrt{k\ell D + 1}\log n + k)(\log n)^2$ and $\eta = 1$. If $\lambda > \ell$ then run the naive random walk algorithm, i.e., the sources find walks of length ℓ simultaneously by sending tokens. Otherwise, do the following. First, modify Phase 2 of SINGLE-RANDOM-WALK to create multiple walks, one at a time; i.e., in the second phase, we stitch the short walks together to get a walk of length ℓ starting at s_1 then do the same thing for s_2, s_3, and so on. We state the theorem below and the proof is in the full version [12].

THEOREM 2.8. MANY-RANDOM-WALKS *finishes in*

$$\tilde{O}\left(\min(\sqrt{k\ell D} + k, k + \ell)\right)$$

rounds with high probability.

3. LOWER BOUND

In this section, we show an almost tight lower bound on the time complexity of performing a distributed random walk. At the end of the walk, we require that each node in the walk should know its correct position(s) among the ℓ steps. We show that any distributed algorithm needs at least $\Omega\left(\sqrt{\frac{\ell}{\log \ell}}\right)$ rounds, even in graphs with low diameter. Note that $\Omega(D)$ is a lower bound [11]. Also note that if a source node wants to sample k destinations from independent random walks, then $\Omega(k)$ is also a lower bound as the source may need to receive $\Omega(k)$ distinct messages. Therefore, for k walks, the lower bound we show is $\Omega(\sqrt{\frac{\ell}{\log \ell}} + k + D)$ rounds. (The rest of the section omits the $\Omega(k+D)$ term.) In particular, we show that there exists a n-node graph of diameter $O(\log n)$ such that any distributed algorithm needs at least $\Omega(\sqrt{\frac{n}{\log n}})$ time to perform a walk of length n. Our lower bound proof makes use of a lower bound for another problem that we call as the *Path Verification problem* defined as follows. Informally, the Path Verification problem is for some node v to verify that a given sequence of nodes in the graph is a valid path of length ℓ.

DEFINITION 3.1 (PATH-VERIFICATION PROBLEM). *The input of the problem consists of an integer ℓ, a graph $G = (V, E)$, and ℓ nodes $v_1, v_2, ..., v_\ell$ in G. To be precise, each node v_i initially has its order number i.*

The goal is for some node v to "verify" that the above sequence of vertices forms an ℓ-length path, i.e., if (v_i, v_{i+1}) forms an edge for all $1 \leq i \leq \ell - 1$. Specifically, v should output "yes" if the sequence forms an ℓ-length path and "no" otherwise.

We show a lower bound for the Path Verification problem that applies to a very general class of verification algorithms defined as follows. Each node can (only) verify a segment of the path that it knows either directly or indirectly (by learning form its neighbors), as follows. Initially each node knows only the trivial segment (i.e. the vertex itself). If a vertex obtains from its neighbor a segment $[i_1, j_1]$ and it has already verified segment $[i_2, j_2]$ that overlaps with $[i_1, j_1]$ (say, $i_1 < i_2 < j_1 < j_2$) then it can verify a larger interval ($[i_1, j_2]$). Note that a node needs to only send the endpoints of the interval that it already verifies (hence larger intervals are better). The goal of the problem is that, in the end, some node verifies the entire segment $[1, \ell]$. We would like to determine a lower bound for the running time of any distributed algorithm for the above problem.

A lower bound for the Path Verification problem, implies a lower bound for the random walk problem as well. The reason is as follows. Both problems involve constructing a path of some specified length ℓ. Intuitively, the former is a simpler problem, since we are not verifying whether the local steps are chosen randomly, but just whether the path is valid and is of length ℓ. On the other hand, any algorithm for the random walk problem (including our algorithm of Section 2), also solves the Path Verification problem, since the path it constructs should be a valid path of length ℓ. It is straightforward to make any distributed algorithm that computes a random walk to also verify that indeed the random walk is a valid walk of appropriate length. This is essential for correctness, as otherwise, an adversary can always change simply one edge of the graph and ensure that the walk is wrong.

In the next section we first prove a lower bound for the Path Verification problem. Then we show the same lower bound holds for the random walk problem by giving a reduction.

3.1 Lower Bound for the Path Verification Problem

The main result of this section is the following theorem.

THEOREM 3.2. *For every n, and $\ell \leq n$ there exists a graph G_n of $\Theta(n)$ vertices and diameter $O(\log n)$, and a path P of length ℓ such that any algorithm that solves the* PATH-VERIFICATION *problem on G_n and P requires more than k rounds, where $k = \sqrt{\frac{\ell}{\log \ell}}$.*

The rest of the section is devoted to proving the above Theorem. We start by defining G_n.

DEFINITION 3.3 (GRAPH G_n). *Let k' be an integer such that k is a power of 2 and $k'/2 \leq 4k < k'$. Let n' be such that $n' \geq n$ and k' divides n'. We construct G_n having $(n' + 2k' - 1) = O(n)$ nodes as follows. First, we construct a path $P = v_1 v_2 ... v_{n'}$. Second, we construct a binary T having k' leaf nodes. Let $u_1, u_2, ..., u_{k'}$ be its leaves from left to right. Finally, we connect P with T by adding an edge $u_i v_{jk'+i}$ for every i and j. We will denote the root of T by x and its left and right children by l and r respectively.*

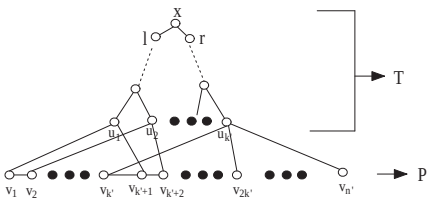

Figure 1: G_n

Clearly, G_n has diameter $O(\log n)$. We then consider a path of length $\ell = \Theta(n)$. If required n can always be made larger by connecting dummy vertices to the root of T. (The resulting graph G_n is as in Figure 1.) □

To prove the theorem, let $\mathcal{A}$ be any algorithm for the PATH-VERIFICATION problem that solves the problem on G_n in at most k' rounds. We need some definitions and claims to prove the theorem.

Definitions of *left/right subtrees* and *breakpoints*..

Consider a tree T' obtained by deleting all edges in P. Notice that nodes $v_{jk'+i}$, for all j and $i \leq k'/2$ are in the subtree of T' rooted at l and all remaining points are in the subtree rooted at r. For any node v, let $sub(v)$ denote the subtree rooted at node v. (Note that $sub(v)$ also include nodes in the path P.) We denote the set of nodes that are leaves of $sub(l)$ by L (i.e., $L = sub(l) \cap P$) and the set of nodes that are leaves in $sub(r)$ by R.

Since we consider an algorithm that takes at most k rounds, consider the situation when the algorithm is given k rounds for *free* to communicate only along the edges of the path P at the beginning. Since L and R consists of every $k'/2$ vertices in P and $k'/2 > 2k$, there are some nodes unreachable from L by walking on P for k steps. In particular, all nodes of the form $v_{jk'+k'/2+k+1}$, for all j, are not reachable from L. We call such nodes *breakpoints* for $sub(l)$. Similarly all nodes of the form $v_{jk'+k+1}$, for all j, are not reachable from R and we call them the breakpoints for $sub(r)$.

Definitions of *path-distance* and *covering*..

For any two nodes u and v in T' (obtained from G_n by deleting edges in P), let $c(u, v)$ be a lowest common ancestor of u and v. We define $path\_dist(u, v)$ to be the number of leaves of subtree of T rooted at $c(u, v)$. Note that the path-distance is defined between any pair of nodes in G_n but the distance is counted using the number of leaves in T (which excludes nodes in P).

We also introduce the notion of the path-distance *covered* by a message. For any message m, the path-distance covered by m is the maximum path-distance taken over all nodes that have held the message m. That is, if m covers some nodes $v'_1, v'_2, ..., v'_k$ then the path-distance covered by m is the number of leaves in the subtrees of T rooted by $v'_1, v'_2, ..., v'_k$. Note that some leaves may be in more than one subtrees and they will be counted only once. Our construction makes the right and left subtrees have a large number of break points, as in the following lemma.

LEMMA 3.4. *The number of breakpoints for the left subtree and for the right subtree are at least $\frac{n}{4k}$ each.*

The reason we define these breakpoints is to show that

the entire information held by the left subtree has many disjoint intervals, and same for the right subtree. This then tells us that the left subtree and the right subtree must *communicate* a lot to be able to merge these intervals by connecting/communicating the break points. To argue this, we show that the total path distance (over all messages) is large, as in the following lemma. (Proof is in the full version [12].)

LEMMA 3.5. *For algorithm $\mathcal{A}$ to solve* PATH-VERIFICATION *problem, the total path-distance covered by all messages is at least n.*

These messages can however be communicated using the tree edges as well. We bound the maximum communication that can be achieved across $sub(l)$ and $sub(r)$ indirectly by bounding the maximum path-distance that can be covered in each round. In particular, we show the following lemma. Proof is in the full version [12].

LEMMA 3.6. *In k rounds, all messages together can cover at most a path-distance of $O(k^2 \log k)$.*

We now describe the proof of the main theorem using these three claims.

PROOF OF THEOREM 3.2. Use Lemmas 3.5 and 3.6 we know that if $\mathcal{A}$ solves PATH-VERIFICATION, then it needs to cover a *path_dist* of n, but in k rounds it can only cover a *path_dist* of $O(k^2 \log k)$. But this is $o(n)$ since $k = \sqrt{\frac{n}{\log n}}$, contradiction. □

3.2 Reduction to Random Walk Problem

We now discuss how the lower bound for the Path Verification problem implies the lower bound of the random walk problem. The main difference between PATH-VERIFICATION problem and the random walk problem is that in the former we can specify which path to verify while the latter problem generates different path each time. We show that the "bad" instance (G_n and P) in the previous section can be modified so that with high probability, the generated random walk is "hard" to verify. The theorems below are stated for ℓ length walk/path instead of n as above. As previously stated, if it is desired that ℓ be $o(n)$, it is always possible to add dummy nodes.

THEOREM 3.7. *For any n, there exists a graph G_n of $\Theta(n)$ vertices and diameter $O(\log n)$, and $\ell = \Theta(n)$ such that, with high probability, a random walk of length ℓ needs $\Omega(\sqrt{\frac{\ell}{\log \ell}})$ rounds.*

PROOF. Theorem 3.2 can be generalized to the case where the path P has infinite capacity, as follows.

THEOREM 3.8. *For any n and $\ell = \Theta(n)$, there exists a graph G_n of $O(n)$ vertices and diameter $O(\log n)$, and a path P of length ℓ such that any algorithm that solves the* PATH-VERIFICATION *problem on G_n and P requires more than $\Omega(\sqrt{\frac{\ell}{\log \ell}})$ rounds, even if edges in P have large capacity (i.e., one can send larger sized messages in one step).*

PROOF. This is because the proof of Theorem 3.2 only uses the congestion of edges in the tree T (imposed above P) to argue about the number of rounds. □

Now, we modify G_n to G'_n as follows. Recall that the path P in G_n has vertices $v_1, v_2, ..., v_{n'}$. For each $i = 1, 2, ..., n'$, we define the weight of an edge (v_i, v_{i+1}) to be $(2n)^{2i}$ (note that weighted graphs are equivalent to unweighted multigraphs in our model). By having more weight, these edges have more capacity as well. However, increasing capacity does not affect the claim as shown above. Observe that, when the walk is at the node v_i, the probability of walk will take the edge (v_i, v_{i+1}) is at least $1 - \frac{1}{n^2}$. Therefore, P is the resulting random walk with probability at least $1 - 1/n$. When the random walk path is P, it takes at least $\sqrt{\frac{n}{\log n}}$ rounds to verify, by Theorem 3.8. This completes the proof. We remark that this construction requires exponential in n number of edges (multiedges). For the distributed computing model, this only translates to a larger bandwidth. The length ℓ is still comparable to the number of nodes.

4. APPLICATIONS

4.1 A Distributed Algorithm for Random Spanning Tree

We now present an algorithm for generating a random spanning tree (RST) of an unweighted undirected network in $\tilde{O}(\sqrt{m}D)$ rounds with high probability. The approach is to simulate Aldous and Broder's [1, 7] RST algorithm which is as follows. First, pick one arbitrary node as a root. Then, perform a random walk from the root node until all nodes are visited. For each non-root node, output the edge that is used for its first visit. (That is, for each non-root node v, if the first time v is visited is t then we output the edge (u, v) where u is the node visited at time $t - 1$.) The output edges clearly form a spanning tree and this spanning tree is shown to come from a uniform distribution among all spanning trees of the graph [1, 7]. The expected time of this algorithm is the expected cover time of the graph which is shown to be $O(mD)$ (in the worst case, i.e., for any undirected, unweighted graph) by Aleniunas et al. [2].

This algorithm can be simulated on the distributed network by our random walk algorithm as follows. The algorithm can be viewed in phases. Initially, we pick a root node arbitrarily and set $\ell = n$. In each phase, we run $\log n$ (different) walks of length ℓ starting from the root node (this takes $\tilde{O}(\sqrt{\ell D})$ rounds using our distributed random walk algorithm). If none of the $O(\log n)$ different walks cover all nodes (this can be easily checked in $O(D)$ time), we double the value of ℓ and start a new phase, i.e., perform again $\log n$ walks of length ℓ. The algorithm continues until one walk of length ℓ covers all nodes. We then use such walk to construct a random spanning tree: As the result of this walk, each node knows its position(s) in the walk (cf. Section 2.2), i.e., it has a list of steps in the walk that it is visited. Therefore, each non-root node can pick an edge that is used in its first visit by communicating to its neighbors. Thus at the end of the algorithm, each node can know which of its adjacent edges belong to the output tree. (An additional $O(n)$ rounds may be used to deliver the resulting tree to a particular node if needed.)

We now analyze the number of rounds in term of τ, the expected cover time of the input graph. The algorithm takes $O(\log \tau)$ phases before $2\tau \leq \ell \leq 4\tau$, and since one of $\log n$ random walks of length 2τ will cover the input graph with high probability, the algorithm will stop with $\ell \leq 4\tau$ with

high probability. Since each phase takes $\tilde{O}(\sqrt{\ell D})$ rounds, the total number of rounds is $\tilde{O}(\sqrt{\tau D})$ with high probability. Since $\tau = \tilde{O}(mD)$, we have the following theorem.

THEOREM 4.1. *The algorithm described above generates a uniform random spanning tree in $\tilde{O}(\sqrt{m}D)$ rounds with high probability.*

4.2 Decentralized Estimation of Mixing Time

We now present an algorithm to estimate the mixing time of a graph from a specified source. Throughout this section, we assume that the graph is connected and non-bipartite (the conditions under which mixing time is well-defined). The main idea in estimating the mixing time is, given a source node, to run many random walks of length ℓ using the approach described in the previous section, and use these to estimate the distribution induced by the ℓ-length random walk. We then compare the distribution at length ℓ, with the stationary distribution to determine if they are *close*, and if not, double ℓ and retry. For this approach, one issue that we need to address is how to compare two distributions with few samples efficiently (a well-studied problem). We introduce some definitions before formalizing our approach and theorem.

DEFINITION 4.2 (DISTRIBUTION VECTOR). *Let $\pi_x(t)$ define the probability distribution vector reached after t steps when the initial distribution starts with probability 1 at node x. Let π denote the stationary distribution vector.*

DEFINITION 4.3. *($\tau^x(\epsilon)$ and τ^x_{mix}, mixing time for source x) Define $\tau^x(\epsilon) = \min t : ||\pi_x(t) - \pi||_1 < \epsilon$. Define $\tau^x_{mix} = \tau^x(1/2e)$.*

The goal is to estimate τ^x_{mix}. Notice that the definition of τ^x_{mix} is consistent due to the following standard monotonicity property of distributions (proof in the full version [12]).

LEMMA 4.4. $||\pi_x(t+1) - \pi||_1 \leq ||\pi_x(t) - \pi||_1$.

To compare two distributions, we use the technique of Batu et. al. [6] to determine if the distributions are ϵ-near. Their result (slightly restated) is summarized in the following theorem.

THEOREM 4.5 ([6]). *For any ϵ, given $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples of a distribution X over $[n]$, and a specified distribution Y, there is a test that outputs PASS with high probability if $|X - Y|_1 \leq \frac{\epsilon^3}{4\sqrt{n}\log n}$, and outputs FAIL with high probability if $|X - Y|_1 \geq 6\epsilon$.*

We now give a very brief description of the algorithm of Batu et. al. [6] to illustrate that it can in fact be simulated on the distributed network efficiently. The algorithm partitions the set of nodes in to buckets based on the steady state probabilities. Each of the $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from X now falls in one of these buckets. Further, the actual count of number of nodes in these buckets for distribution Y are counted. The exact count for Y for at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ buckets (corresponding to the samples) is compared with the number of samples from X; these are compared to determine if X and Y are close. We refer the reader to their paper [6] for a precise description.

Our algorithm starts with $\ell = 1$ and runs $K = \tilde{O}(\sqrt{n})$ walks of length ℓ from the specified source x. As the test of comparison with the steady state distribution outputs FAIL (for choice of $\epsilon = 1/12e$), ℓ is doubled. This process is repeated to identify the largest ℓ such that the test outputs FAIL with high probability and the smallest ℓ such that the test outputs PASS with high probability. These give lower and upper bounds on the required τ^x_{mix} respectively. Our resulting theorem is presented below and the proof is in the full version [12].

THEOREM 4.6. *Given a graph with diameter D, a node x can find, in $\tilde{O}(n^{1/2} + n^{1/4}\sqrt{D\tau^x(\epsilon)})$ rounds, a time $\tilde{\tau}^x_{mix}$ such that $\tau^x_{mix} \leq \tilde{\tau}^x_{mix} \leq \tau^x(\epsilon)$, where $\epsilon = \frac{1}{6912e\sqrt{n}\log n}$.*

PROOF. For undirected unweighted graphs, the stationary distribution of the random walk is known and is $\frac{deg(i)}{2m}$ for node i with degree $deg(i)$, where m is the number of edges in the graph. If a source node in the network knows the degree distribution, we only need $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples from a distribution to compare it to the stationary distribution. This can be achieved by running MULTIPLERANDOMWALK to obtain $K = \tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ random walks. We choose $\epsilon = 1/12e$. To find the approximate mixing time, we try out increasing values of l that are powers of 2. Once we find the right consecutive powers of 2, the monotonicity property admits a binary search to determine the exact value for the specified ϵ.

The result in [6] can also be adapted to compare with the steady state distribution even if the source does not know the entire distribution. As described previously, the source only needs to know the *count* of number of nodes with steady state distribution in given buckets. Specifically, the buckets of interest are at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ as the count is required only for buckets were a sample is drawn from. Since each node knows its own steady state probability (determined just by its degree), the source can broadcast a specific bucket information and recover, in $O(D)$ steps, the count of number of nodes that fall into this bucket. Using the standard upcast technique previously described, the source can obtain the bucket count for each of these at most $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ buckets in $\tilde{O}(n^{1/2}poly(\epsilon^{-1}) + D)$ rounds.

We have shown previously that a source node can obtain K samples from K independent random walks of length ℓ in $\tilde{O}(K + \sqrt{Kl}D)$ rounds. Setting $K = \tilde{O}(n^{1/2}poly(\epsilon^{-1}) + D)$ completes the proof. □

Suppose our estimate of τ^x_{mix} is close to the mixing time of the graph defined as $\tau_{mix} = \max_x \tau^x_{mix}$, then this would allow us to estimate several related quantities. Given a mixing time τ_{mix}, we can approximate the spectral gap $(1 - \lambda_2)$ and the conductance (Φ) due to the known relations that $\frac{1}{1-\lambda_2} \leq \tau_{mix} \leq \frac{\log n}{1-\lambda_2}$ and $\Theta(1 - \lambda_2) \leq \Phi \leq \Theta(\sqrt{1 - \lambda_2})$ as shown in [20].

5. CONCLUDING REMARKS

This paper makes progress towards resolving the time complexity of distributed computation of random walks in undirected networks. The dependence on the diameter D is still not tight, and it would be interesting to settle this. There is also a gap in our bounds for performing k independent random walks. Further, we look at the CONGEST model enforcing a bandwidth restriction and minimize number of rounds. While our algorithms have good *amortized* message complexity over several walks, it would be nice to

come up with algorithms that are round efficient and yet have smaller message complexity.

We presented two algorithmic applications of our distributed random walk algorithm: estimating mixing times and computing random spanning trees. It would be interesting to improve upon these results. For example, is there a $\tilde{O}(\sqrt{\tau^x_{mix}}+n^{1/4})$ round algorithm to estimate τ^x; and is there a $\tilde{O}(n)$ round algorithm for RST?

There are several interesting directions to take this work further. Can these techniques be useful for estimating the second eigenvector of the transition matrix (useful for sparse cuts)? Are there efficient distributed algorithms for random walks in directed graphs (useful for PageRank and related quantities)? Finally, from a practical standpoint, it is important to develop algorithms that are robust to failures and it would be nice to extend our techniques to handle such node/edge failures.

6. REFERENCES

[1] D. Aldous. The random walk construction of uniform spanning trees and uniform labelled trees. *SIAM J. Discrete Math.*, 3(4):450–465, 1990.

[2] R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *FOCS*, 1979.

[3] N. Alon, C. Avin, M. Koucký, G. Kozma, Z. Lotker, and M. R. Tuttle. Many random walks are faster than one. In *SPAA*, pages 119–128, 2008.

[4] H. Baala, O. Flauzac, J. Gaber, M. Bui, and T. A. El-Ghazawi. A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *J. Parallel Distrib. Comput.*, 63(1):97–104, 2003.

[5] J. Bar-Ilan and D. Zernik. Random leaders and random spanning trees. In *3rd International Workshop on Distributed Algorithms (later called DISC)*, 1989.

[6] T. Batu, L. Fortnow, E. Fischer, R. Kumar, R. Rubinfeld, and P. White. Testing random variables for independence and identity. In *FOCS*, pages 442–451, 2001.

[7] A. Broder. Generating random spanning trees. In *FOCS*, 1989.

[8] M. Bui, T. Bernard, D. Sohier, and A. Bui. Random walks in distributed computing: A survey. In *IICS*, pages 1–14, 2004.

[9] C. Cooper, A. Frieze, and T. Radzik. Multiple random walks in random regular graphs. In *Preprint*, 2009.

[10] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. In *PODS*, pages 69–78, 2008.

[11] A. Das Sarma, D. Nanongkai, and G. Pandurangan. Fast distributed random walks. In *PODC*, pages 161–170, 2009.

[12] A. Das Sarma, D. Nanongkai, G. Pandurangan, and P. Tetali. Efficient distributed random walks with applications. *CoRR*, abs/0911.3195, 2010. URL: `http://arxiv.org/abs/0911.3195`.

[13] D. Dubhashi, F. Grandioni, and A. Panconesi. Distributed algorithms via lp duality and randomization. In *Handbook of Approximation Algorithms and Metaheuristics*. 2007.

[14] M. Elkin. An overview of distributed approximation. *ACM SIGACT News Distributed Computing Column*, 35(4):40–57, December 2004.

[15] M. Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM J. Comput.*, 36(2):433–456, 2006. Also appeared in STOC'04.

[16] R. Elsässer and T. Sauerwald. Tight bounds for the cover time of multiple random walks. In *ICALP (1)*, pages 415–426, 2009.

[17] J. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27:302–316, 1998.

[18] C. Gkantsidis, G. Goel, M. Mihail, and A. Saberi. Towards topology aware networks. In *IEEE INFOCOM*, 2007.

[19] N. Goyal, L. Rademacher, and S. Vempala. Expanders via random spanning trees. In *SODA*, 2009.

[20] M. Jerrum and A. Sinclair. Approximating the permanent. *SIAM Journal of Computing*, 18(6):1149–1178, 1989.

[21] J. Kelner and A. Madry. Faster generation of random spanning trees. In *IEEE FOCS*, 2009.

[22] D. Kempe and F. McSherry. A decentralized algorithm for spectral analysis. *Journal of Computer and System Sciences*, 74(1):70–83, 2008.

[23] M. Khan, F. Kuhn, D. Malkhi, G. Pandurangan, and K. Talwar. Efficient distributed approximation algorithms via probabilistic tree embeddings. In *PODC*, pages 263–272, 2008.

[24] M. Khan and G. Pandurangan. A fast distributed approximation algorithm for minimum spanning trees. *Distributed Computing*, 20:391–402, 2008.

[25] S. Kutten and D. Peleg. Fast distributed construction of k-dominating sets and applications. *J. Algorithms*, 28:40–66, 1998.

[26] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[27] S. Muthukrishnan and G. Pandurangan. The bin-covering technique for thresholding random geometric graph properties. In *ACM SODA*, 2005. To appear in Journal of Computer and System Sciences.

[28] G. Pandurangan and M. Khan. Theory of communication networks. In *Algorithms and Theory of Computation Handbook, Second Edition*. CRC Press, 2009.

[29] D. Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[30] D. Peleg and V. Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, 2000. Also in FOCS'99.

[31] R. Sami and A. Twigg. Lower bounds for distributed markov chain problems. *CoRR*, abs/0810.5263, 2008.

[32] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985. Also in FOCS'83.

[33] M. Zhong and K. Shen. Random walk based node sampling in self-organizing networks. *Operating Systems Review*, 40(3):49–55, 2006.