Estimating PageRank on Graph Streams

Atish Das Sarma
Georgia Tech.
atish@cc.gatech.edu

Sreenivas Gollapudi
Microsoft Research
sreenig@microsoft.com

Rina Panigrahy
Microsoft Research
rina@microsoft.com

ABSTRACT

This study focuses on computations on large graphs (e.g., the web-graph) where the edges of the graph are presented as a stream. The objective in the streaming model is to use small amount of memory (preferably sub-linear in the number of nodes n) and a few passes.

In the streaming model, we show how to perform several graph computations including estimating the probability distribution after a random walk of length l, mixing time, and the conductance. We estimate the mixing time M of a random walk in $\tilde{O}(n\alpha + M\alpha\sqrt{n} + \sqrt{\frac{Mn}{\alpha}})$ space and $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes. Furthermore, the relation between mixing time and conductance gives us an estimate for the conductance of the graph. By applying our algorithm for computing probability distribution on the web-graph, we can estimate the *PageRank* p of any node up to an additive error of $\sqrt{\epsilon p}$ in $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes and $\tilde{O}(\min(n\alpha + \frac{1}{\epsilon}\sqrt{\frac{M}{\alpha}} + \frac{1}{\epsilon}M\alpha, \alpha n\sqrt{M\alpha} + \frac{1}{\epsilon}\sqrt{\frac{M}{\alpha}}))$ space, for any $\alpha \in (0, 1]$. In particular, for $\epsilon = M/n$, by setting $\alpha = M^{-\frac{1}{2}}$, we can compute the approximate PageRank values in $\tilde{O}(nM^{-\frac{1}{4}})$ space and $\tilde{O}(M^{\frac{3}{4}})$ passes. In comparison, a standard implementation of the PageRank algorithm will take $O(n)$ space and $O(M)$ passes.

Categories and Subject Descriptors

F.2.3 [**Theory of Computation**]: Analysis of algorithms and problem complexityTradeoffs among Complexity Measures

General Terms

Algorithms, Theory, Performance

Keywords

Graph conductance, Mixing Time, PageRank, Random Walk, Streaming Algorithms

ACM PODS '08 Vancouver, BC, Canada

1. INTRODUCTION

The scale of the Internet has greatly increased both our ability to produce and the need to process extremely large amounts of data. While most data can be stored in secondary storage, processing such large data is usually performed using physical memory (RAM) which is a limited resource. The streaming model admits an ideal approach to processing the data. In this model, the data is presented as a stream and any computation on the stream relies on using a small amount of memory. Many streaming algorithms exist for computing frequency moments (with matching lower bounds), quantiles, and norms [1, 21, 7, 6, 25, 15, 16], and computations over graphs including counting triangles, properties of degree sequences, and connectivity [4, 10, 11, 13, 24, 26]. In this study we compute the PageRank [8] of a large graph presented as a stream of edges in no particular order; neither is it required that all the edges incident on a vertex be grouped together in the stream. Real world networks such as the web and social networks can be modeled as large graphs. Other instances of large graphs include click graphs generated from search engine query logs, document-term graphs computed from large collection of documents etc. These graphs described above readily admit a streaming model. Moreover, they also admit link-based ranking algorithms like the PageRank algorithm to compute the relative importance of nodes in the graph.

While the basic requirements of streaming algorithms include small space and a small number of passes, these quantities can vary significantly from algorithm to algorithm. Demetrescu et. al. [11] give an excellent exposition on the space-passes trade off in graph streaming problems. After Henzinger et. al. [20] showed linear lower bounds on the "space × passes" product for several problems including connectivity and shortest path problems, there has been the work of Demetrescu et. al. [11] that proposed streaming graph algorithms using sublinear space and passes to compute shortest paths on directed graphs. In this study we propose algorithms that require sublinear space and passes to compute the approximate PageRank values of nodes in a large directed graph.

The graph is stored in secondary storage as its large size makes it infeasible to store the entire graph in main memory. Hence, it is only practical to process such graphs with a small amount of memory even at the expense of using multiple passes. Given a web-graph representing the web pages and links between them, the PageRank algorithm computes the probability distribution of a random surfer visiting a page on the web. In general the PageRank of a page u is dependent

on the PageRank of all pages v that link to u as $PR(u) = \sum_{(v,u)\in E} PR(v)/d(v)$ where $d(.)$ represents the out-degree. A standard implementation of the algorithm requires several iterations, say M, before the values converge. Alternately, this process can be viewed as a random walk requiring M steps. The typical length is about 200. In fact, the random walk is performed on a slightly modified graph that captures a random reset step in the PageRank algorithm. This step was introduced to model the random jump of a surfer from a page with no out-links to another page on the web and it also improves the convergence time of the algorithm. Typically, in a PageRank computation, the nodes with large PageRank are of interest.

Besides PageRank, other graph properties of interest include connectedness, conductance, mixing time, and the sparsest cut. It is well known and was first shown by Jerrum and Sinclair [23] that the mixing time M and conductance ϕ are related as $\phi^2/2 \leq \frac{1}{M} \leq 2\phi$ [1].

1.1 Contributions of this study

A random walk of length l can be modeled as a matrix-vector computation $v = A^l u$, where u is the initial distribution on the nodes (typically uniform), A is the transition matrix that corresponds to a single step in the random walk on the underlying graph, and v is the final distribution after performing the walk. The problem of computing a single destination of a random walk of length l starting from a node picked from the distribution u is same as sampling a node from the distribution v. So by simply maintaining an array of size n that represents the probability distribution, we can compute v in l passes and $O(n)$ space. Thus, a standard implementation of the PageRank algorithm, that computes the stationary distribution, will require M passes and $O(n)$ space, where M is the mixing time. In comparison, our work requires $\tilde{O}(\sqrt{M})$ passes and $o(n)$ space to compute the PageRank of nodes with values greater than M/n. This requires the knowledge of the mixing time. We also provide an algorithm to estimate the mixing time. In this paper we provide algorithms on a graph stream for the following problems -

- Running a single random walk of length l in $O(\sqrt{l})$ passes. In fact, we show how to perform n/l independent random walks using space sublinear in n and passes sublinear in l,
- Approximating the probability distribution (the PageRank vector) of nodes after a random walk of length l.
- Approximating the mixing time, i.e., the time taken before the random walk starting from an initial distribution reaches within ϵ of the steady state distribution under the L_1-norm.
- Estimating the conductance of a graph.

For all these results, the main goal is to use as few passes over the stream as possible, while using space sub-linear in the number of nodes in the graph. Notice that for a dense graph (number of edges $\Omega(n^2)$), the space used by our algorithms are asymptotically less than square-root of the length of the stream. To compute the probability distribution over nodes after a random walk of length l, a naive algorithm uses l passes and $O(n)$ space by performing l matrix-vector multiplications. We show how to approximate the same distribution; For any node with probability p in the distribution, we can approximate p within $p \pm \sqrt{\epsilon p}$ in $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes and $\tilde{O}(\min(n\alpha + \frac{1}{\epsilon}\sqrt{\frac{l}{\alpha}} + \frac{1}{\epsilon} l\alpha, \alpha n\sqrt{l\alpha} + \frac{1}{\epsilon}\sqrt{\frac{l}{\alpha}}))$ space. Note that approximating p within $p \pm \sqrt{\epsilon p}$ can also be viewed as a $1 \pm \sqrt{\epsilon/p}$ approximation ratio which is close to 1 for p much larger than ϵ. This means we can estimate the probability value with high accuracy for nodes with large probability. Note that in the context of PageRank computation, we set l to the mixing time M of the random walk. For concreteness, this means we can estimate the PageRank p of a node within $p \pm \sqrt{\epsilon p}$ in $\tilde{O}(nM^{-\frac{1}{4}})$ space and $\tilde{O}(M^{\frac{3}{4}})$ passes for $\epsilon = M/n$. We also show how to find what we call the ϵ-near mixing time, i.e. the time taken for the probability distribution to reach within ϵ of the steady state distribution under the L_1-norm. This algorithm takes $\tilde{O}(n\alpha + poly(\epsilon^{-1})(M\alpha\sqrt{n} + \sqrt{\frac{Mn}{\alpha}}))$ space and $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes on undirected graphs, where M is the actual mixing time of the graph. This automatically gives a result for estimating the *conductance* of the graph.

1.2 Related Work

Data streaming algorithms became popular since the famous result of Alon et. al. [1] on approximating frequency moments. Since then, there has been a surge of papers looking at various problems in the data streams setting. In particular, there has been significant attention to computing various frequency moments as they provide important statistics about the data stream. Tight results are known for computing some of them, while others remain open [7, 21]. Recent streaming results include estimating earth-movers distance [22], communication problems [14, 30], counting triangles in graphs [4, 24, 9], quantile estimation [16, 19], sampling and entropy information [18, 17], and graph matchings [26].

This is by no means a comprehensive summary of all the results on data streams. There are many more studies on streaming problems including filtering irrelevant data from the stream, low rank approximation and fast multiplication of matrices, characterizing sketchable distances, scheduling problems, work on dynamic geometric problems, generating histograms, and finding long increasing subsequences in data streams.

In comparison to the work on aggregating statistics of a data stream, the work on graphs as data streams is limited. Demetrescu et.al. [11] show space-pass trade-offs for shortest path problems in graph streams. In general, it seems hard to approximate many properties on graphs while maintaining sub-linear space in the number of vertices in the graph, by performing only a constant passes over the stream.

A very interesting piece of work is due to Sarlos *et al* [28] who give approaches to finding summaries for hyperlink analysis and personalized PageRank computation. Their study is not under the data streams setting; rather, they use sketching techniques and construct simple deterministic summaries that are later used by the algorithms for computing the PageRank vectors. They give lower bounds to prove that

[1] Note that mixing time is well-defined only for *aperiodic* graphs. Bipartite graphs, for example, are not aperiodic. On the other hand, the bipartiteness of a graph can be checked in a single pass, see e.g., [13]

the space required by their algorithms is *optimal*, under their setting. Given an additive error of ϵ and the probability δ of an incorrect result, their disk usage bound is $O(n \log(1/\delta)/\epsilon)$.

There has also been work on s, t connectivity [12, 3] not under the streams setting; however it is not clear that extending to the streaming model is possible. Recent work by Wicks, Greenwald [29] shows an interesting approach to parallelizing the computation of PageRank. McSherry [27] exploits the link structure of the web graph within and across domains to accelerate the computation of PageRank. Andersen et. al. [2] computes local cuts near a specified vertex using personalized PageRank vectors

The main ingredient in all our algorithms is to perform random walks efficiently. We begin by presenting the algorithm for running one random walk of length l in a small number of passes, in Section 2. The main idea in this algorithm is to sample each of the n nodes independently with probability α and perform short (length w) random walks from each of the sampled nodes, in w passes. We then try to *merge* these walks to form longer random walks from the source. The main idea in estimating the probability distribution or mixing time is running several such random walks. However, running several random walks requires some additional ideas to ensure small space. We describe how to efficiently run a large number of random walks in Section 3. This also gives an algorithm for approximating the probability distribution after a random walk. The algorithm for approximating the mixing time uses these ideas and is described in Sections 4. Section 5 provides an alternate algorithm for estimating the probability distribution with higher accuracy more efficiently for certain values of ϵ.

2. SINGLE RANDOM WALK

We first present an algorithm to perform a single random walk over a graph stream efficiently. The naive approach is to do this in $O(1)$ space and l passes by performing one step of the random walk with every pass over the stream. In a randomized edge stream (as opposed to an arbitrary edge stream), one can do slightly more than one step with every pass, in expectation. However, this still requires $O(l)$ passes over the stream for a random walk of length l. At the other extreme, one can perform a random walk of length l in 1 pass and $O(nl)$ space by sampling l edges out of each of the n nodes in one pass. Subsequently, with these nl edges stored, it is possible to perform a random walk of length l without any more passes, as with l edges out of each node, the random walk cannot get stuck at a node before completing a walk of length l. In this section, we show the following result.

THEOREM 2.1. *One can perform a random walk of length l in $O(\sqrt{\frac{l}{\alpha}})$ passes and $O(n\alpha + \sqrt{\frac{l}{\alpha}})$ space, for any choice of α with $0 < \alpha \leq 1$.*

Setting $\alpha = 1$, we get the following corollary.

COROLLARY 2.2. *One can perform a random walk of length l in $O(\sqrt{l})$ passes and $O(n)$ space.*

We first describe the overall approach of our algorithm.

Perform short random walks out of sampled nodes - The main idea in our algorithm is to sample each node with probability α independently and perform short random walks of length w from each sampled node; this is done in w passes over the stream. The algorithm tries to extend the walk from the source by *merging* these short walks to form a longer random walk. It may get stuck in one of two ways. First, the walk may end up at a node that has not been sampled. Second, the walk may end up at one of the sampled nodes for a second time; notice that its stored w length walk cannot be used more than once in order to preserve the randomness of the walk. Note that sampling each node with probability α can be done by using a pseudo-random hash function on the node id.

Handling stuck nodes - While constructing the walk if it gets stuck at a node, from which no unused w-length walk is available, we will refer to such a node as a *stuck* node. We handle stuck nodes as follows. We keep track of the set S of sampled nodes whose w length walks have already been used in extending the random walk so far. We sample s edges out of the stuck node and each node in S in one pass. We then extend the walk as far as possible using these newly sampled edges. If the new end-point is a sampled node whose w-length walk has not been used (i.e., it is not in S), then we continue merging as before. Otherwise, if the new end-point is a new stuck node, we repeat the process of sampling s edges out of S and all the stuck nodes visited since the last w-length walk was used. Finally, if the new end-point is not a stuck node, we continue appending w length walks as before.

We need to argue that whenever the algorithm hits a stuck node, it makes sufficient progress with each pass. Note that after each round of sampling s edges out of the stuck nodes and the nodes in S, either we reach a node that is not stuck and hence make w progress, or we make a progress of s steps and find no new stuck node or we find a new stuck node. The point is that the we cannot keep finding new stuck nodes repeatedly for too long as each new node is a sampled node with probability α. So, it is unlikely we will visit more than $O(1/\alpha)$ new stuck nodes in a sequence before becoming unstuck.

The notation in the algorithm SINGLERANDOMWALK uses T to denote the sampled nodes obtained by sampling each node with probability α independently. The table W indexed by a sampled node (say t) stores the end point of the w length walks starting at t as $W[t]$. Note that this table can be populated in w passes while using $O(\alpha n)$ space. The set S keeps track of all nodes in T whose w length walks get used up. The algorithm continues extending the walk using the w length walks implicitly stored in the table W until it finds a stuck node. The module HANDLESTUCKNODE proceeds by sampling s edges out of $S \cup R$ where R is the set of stuck nodes visited in the current invocation.

REMARK 2.3. *The length of the walk produced by algoritm* SINGLERANDOMWALK *could exceed l slightly (by at most w). To prevent this we can run the algorithm till we get a walk of length at least $l - w$ and then extend this walk to length l in at most w additional passes.*

REMARK 2.4. *Note that since we only store the end-points of w length walks in W, the internal nodes are not available in $\mathcal{L}_u$ at the end of the algorithm* SINGLERANDOMWALK. *These w-length walks can be reconstructed by making pseudo-random choices while creating the w length random walks in Step 3 of* SINGLERANDOMWALK, *and reusing the coin tosses to reconstruct them at the end. A single pseudo-random hash*

Algorithm 1 SINGLERANDOMWALK(u, l)

1: **Input:** Starting node u, and desired walk length l.
2: **Output:** $\mathcal{L}_u$ the random walk from u of length l.
3: $T \leftarrow$ set of nodes obtained by sampling each node independently with probability α (in one pass).
4: In w passes, perform walks of length w from every node in T. Let $W[t] \leftarrow$ the end point of the walk of length w from $t \in T$ (the nodes in T whose w length walks get used towards $\mathcal{L}_u$ will get included in S).
5: $S \leftarrow \{\}$(we will refer to the nodes in S as *centers*).
6: Initialize $\mathcal{L}_u$ to a zero length walk starting at u. Let $x \leftarrow u$.
7: **while** $|\mathcal{L}_u| < l$ **do**
   1. if ($x \in T$ and $x \notin S$) extend $\mathcal{L}_u$ by appending the walk (implicit in) $W[x]$. $S \leftarrow S \cup \{x\}$. $x \leftarrow W[x]$, the new end point of $\mathcal{L}_u$. {this means we have a w length walk starting at x that has not been used so far in $\mathcal{L}_u$}
   2. if ($x \notin T$ or $x \in S$) HANDLESTUCKNODE(x, T, S, $\mathcal{L}_u$, l). {this means that either x was not in the initial set of sampled nodes, or x's w-length walk has already been used up}
8: **end while**

Algorithm 2 HANDLESTUCKNODE(x, T, S, $\mathcal{L}_u$, l)

1: $R \leftarrow x$.
2: **while** $|\mathcal{L}_u| < l$ **do**
3: $E \leftarrow$ sample s edges (with repetition) out of each node in $S \cup R$.
4: Extend $\mathcal{L}_u$ as far as possible by walking along the sampled edges in E (on visiting a node in $S \cup R$ for the k-th time, use the k-th edge of the s sampled edges from that node).
5: $x \leftarrow$ new end point of $\mathcal{L}_u$ after the extenstion. One of the following cases arise.
   1. if ($x \in S \cup R$) **continue** {no new node is seen, at least s progress has been made.}
   2. if ($x \in T$ and $x \notin S \cup R$) **return** {this means that x is a node that has not been seen in the walk so far, and x was among the set of nodes sampled initially; therefore, the w-length walk from x has not been used}
   3. if ($x \notin T$ and $x \notin S \cup R$) $R \leftarrow R \cup \{x\}$. {this means that x is a new node that has not been visited in this invocation, and x is not in the initial set sampled nodes T}
6: **end while**

function can be used to generate all the coin tosses.

We begin the analysis with a lemma that follows immediately from the algorithm.

LEMMA 2.5. $|S| \leq \frac{l}{w}$.

PROOF. A node is added to the set S only after we use a w length walk from one of the sampled nodes. If we perform a walk of length l, we will end up using at most $\frac{l}{w}$ walks of length w. □

We now state and prove the main claim that is required in bounding the number of passes required by algorithm SINGLERANDOMWALK in performing a random walk of length l.

CLAIM 2.6. *With every additional pass over the edge stream (after the first w passes), the length of the random walk $\mathcal{L}_u$ increases by at least $O(\min\{s, \alpha w\})$ in amortization.*

PROOF. We only need to examine the algorithm HANDLESTUCKNODE. An additional pass over the stream is made when s edges are sampled from every node in $S \cup R$. This happens when the algorithm gets stuck at a new stuck node in R. After a pass over the stream, either the algorithm makes s progress, or a *new node* is visited. In the latter case, with probability α, the new node is in T (since T contains each node with probability α), and with probability $1 - \alpha$, it is a new stuck node. If the new node is not a stuck node, w progress is made. Since the probability of not seeing a new node in T is $1 - \alpha$ with every additional pass, the probability that more than $O(1/\alpha)$ new stuck nodes are seen before a new node in T is seen is small, by Chernoff Bounds. So w.h.p., $|R|$ is less than $O(1/\alpha)$ in each invocation of HANDLESTUCKNODE. Therefore the number of passes in which s progress is not made, is no more than $O(1/\alpha)$, at the end of which w progress is made giving $\frac{w}{O(1/\alpha)}$ average progress per pass. By this amortized argument, the walk makes $O(\min\{s, \alpha w\})$ progress with every pass over the edge stream. Furthermore, w.h.p., in r passes, the algorithm SINGLERANDOMWALK makes a progress of at least $O(r \cdot \min\{s, \alpha w\})$ (for $r \geq \Omega(1/\alpha)$). □

We are now ready to prove Theorem 2.1.

PROOF OF THEOREM 2.1. *Correctness:* We first argue that the walk of length l from source u generated by our algorithm is indeed a *random* walk. Notice that the algorithm uses each w-length walk only once in the walk $\mathcal{L}_u$. Since the algorithm never reuses any randomly sampled edges or walks, and whenever we sample s edges, we pick the ith sampled edge, when visiting the node for the ith time, randomness is maintained. It is important to note that while sampling s edges, we (correctly) allow the same edge to be sampled multiple times; in particular, this would definitely happen for a vertex with degree less than s.

Space: We need space $O(\alpha n)$ for storing the sampled nodes and the end point of their w length walks. Using Lemma 2.5, sampling s edges from every node in S requires $O(s\frac{l}{w})$ space, while sampling s edges from the nodes not in R takes up an additional $O(s\frac{1}{\alpha})$ space as $|R| = O(1/\alpha)$. It follows, the total space used in algorithm SINGLERANDOMWALK is $O(n\alpha + s(\frac{l}{w} + \frac{1}{\alpha}))$.

Passes: The most crucial observation in analyzing the number of passes required by the algorithm SINGLERANDOMWALK is Claim 2.6. This claim states that, in r passes over the stream (after the first w passes), for $r > \Omega(1/\alpha)$, with high probability, the random walk $\mathcal{L}_u$ makes a progress of at least $r \cdot O(\min\{s, w\alpha\})$.

Additionally, w passes are used for generating the w length walks from each of the sampled $O(\alpha n)$ nodes. Therefore, for completing the walk of length l, the number of passes required, apart from the first $w+1$ passes (for sampling nodes and generating w length walks from them) is $O(\frac{l}{\min\{s,\alpha w\}}) = O(\frac{l}{s} + \frac{l}{\alpha w})$. It follows, the total number of passes used in the algorithm is $O(w + \frac{l}{s} + \frac{l}{w\alpha})$.

Setting $s = \sqrt{l\alpha}$ and $w = \sqrt{\frac{l}{\alpha}}$ completes the proof. □

Note that SINGLERANDOMWALK takes sublinear space and passes even for performing very long ($O(n)$ length) random walks. Setting $l = O(n)$ and choosing $\alpha = n^{-\frac{1}{3}}$ in Theorem 2.1 gives the following corollary.

COROLLARY 2.7. *One can perform a random walk of length $O(n)$ in $O(n^{2/3})$ passes and $O(n^{2/3})$ space.*

The above algorithm can easily be extended to the case when the starting node of the random walk comes from a distribution, rather than a specific node. In this case, one can sample a node from the initial distribution and use this as the source node for the random walk.

Notice that if we wanted to perform a larger number of independent random walks using this algorithm directly, the space required would increase linearly in the number of walks, while the passes would remain unchanged. The bottle-neck in the space requirement would arise due to two reasons. First, the algorithm would need to store multiple w-length walks from each sampled node, one for each random walk. Second, many of these random walks could get stuck at the same time, and the algorithm may be required to sample s edges out of the centers of many walks. In the following section, we reduce the space requirements arising in both these scenarios by trying to identify the *appropriate* number of w-length walks required for each sampled node.

3. ESTIMATING PROBABILITY DISTRIBUTION BY PERFORMING A LARGE NUMBER OF RANDOM WALKS

We now show how to estimate the probability distribution of the destination node after performing a random walk. We achieve this by performing several random walks. The source node may either be fixed or chosen from a certain initial distribution. A naive method that uses algorithm SINGLERANDOMWALK to perform K random walks would require $O(K(n\alpha + \sqrt{\frac{l}{\alpha}}))$ space. In this section, we show how algorithm SINGLERANDOMWALK can be extended to perform n/l random walks without significant increase in the space-pass complexity. Specifically, we show the following result.

THEOREM 3.1. *One can perform K random walks of length l in $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes and $\tilde{O}(n\alpha + K\sqrt{\frac{l}{\alpha}} + Kl\alpha)$ space for any choice of α with $0 < \alpha \leq 1$.*

In particular, if $K = \frac{n}{l}$, then for $\alpha \geq l^{-1/3}$, the space requirement is $O(n\alpha)$ which is as good as the space complexity of SINGLERANDOMWALK. We first describe how Theorem 3.1 can be used to estimate the probability distribution after a random walk of length l.

By performing a large number of random walks and computing the fraction of walks that end at a given node gives us an estimate of the probability that a random walk ends at this node. If the actual probability of ending at a node is p, then by setting $K = \frac{\log n}{\epsilon}$, we get an estimate for p with accuracy $p \pm \sqrt{\epsilon p}$. By Chernoff bounds, due to the $\log n$ factor, this estimate is valid w.h.p. for all nodes.

COROLLARY 3.2. *For any node with probability p in the probability distribution after a walk of length l, one can approximate its probability up to an accuracy of $p \pm \sqrt{p\epsilon}$ in $\tilde{O}(\sqrt{\frac{l}{\alpha}})$ passes and $\tilde{O}(n\alpha + \frac{1}{\epsilon}\sqrt{\frac{l}{\alpha}} + \frac{1}{\epsilon}l\alpha)$ space for any choice of α with $0 < \alpha \leq 1$. This is a $(1 \pm \sqrt{\frac{\epsilon}{p}})$-factor approximation for a node with probability p in the distribution.*

Notice that for $p \gg \epsilon$, this is a constant factor approximation close to 1.

By applying this algorithm on the web-graph, we can estimate the PageRank vector up to an accuracy of $p \pm \sqrt{p\epsilon}$ for any node with probability p in the stationary distribution, in $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes and $\tilde{O}(n\alpha + \frac{1}{\epsilon}\sqrt{\frac{M}{\alpha}} + \frac{1}{\epsilon}M\alpha)$ space, where M is the mixing time of the graph. We will show later in Section 4 how to estimate the mixing time M of any graph. Note that our algorithm is also able to handle the random resets in the standard definition of PageRank by handling these transitions implicitly; that is the transition edges corresponding to the random resets can be included into the graph implicitly.

Our technique for performing a large number of walks uses algorithm SINGLERANDOMWALK as a subroutine. The key idea in our approach is to *estimate* the probability p_i that the w length walk of node i is used in SINGLERANDOMWALK. We then use the p_i's to store the appropriate number of w length walks from each sampled node for K executions of SINGLERANDOMWALK. Estimating p_i however again requires performing random walks. For this purpose, we start by performing one random walk, then two, then four and so on. Doubling the number of random walks in every phase, we get more and more accurate estimates of p_i for the sampled nodes.

Let us first define p_i for every node i assuming a given set of sampled nodes in Algorithm SINGLERANDOMWALK.

DEFINITION 3.3. *For any sampled node i, define p_i to be the probability that on running the algorithm* SINGLERANDOMWALK*, the w length walk of node i is used (and hence i gets included in the set of centers S in the performed walk of length l).*

Lemma 2.5 states that $|S| \leq l/w$. Notice that a node is included in S if and only if its w length random walk is used towards the random walk $\mathcal{L}_u$. By our definition of p_i, we have that $\sum_i p_i$ is equal to the expected size of $|S|$. From these two statements, we get the following observation.

OBSERVATION. $\sum_i p_i \leq \frac{l}{w}$.

We now describe the algorithm for performing a large number of random walks. Whenever we say *sample x walks of length w from i*, we mean take the end-points of x independent random walks of length w starting at i.

This algorithm runs in phases. To obtain K walks of length l, algorithm MULTIPLERANDOMWALK is run for $j = \log K$ phases. In phase $j+1$ we run $O(2^j \log n)$ parallel executions of SINGLERANDOMWALK and use these to estimate the p_i to an additive error of $\sqrt{p_i/2^j}$. This estimate is then used in the next phase to store the appropriate number of w length walks from each i. Note that, all the executions of SINGLERANDOMWALK share the same set of sampled nodes.

Algorithm 3 MULTIPLERANDOMWALK($\mathcal{I}$, l, K)

1: **Input**: Distribution of the source nodes, $\mathcal{I}$, and length of the walk, l
2: $T \leftarrow$ set of nodes obtained by sampling each node independently with probability α.
3: Perform phases 1 through $\log K$ as follows -
4: **Phase 1:**
   1. Perform $O(\log n)$ walks of length w from each of the sampled nodes, in w passes. This are sufficient to simulate the Step 1 of $O(\log n)$ parallel executions of SINGLERANDOMWALK.
   2. Spawn $K_1 = O(\log n)$ instances of SINGLERANDOMWALK to obtain K_1 walks. All these instances use only the w-length walks in the previous step.
   3. For each sampled node $i \in T$, estimate $p_i = n_i/K_1$ where n_i is the number of walks produced in the previous step that use i as a center. w.h.p., this estimate is accurate for nodes with actual $p_i \geq \frac{1}{2}$, by Chernoff bounds.
5: **Phase** $(j+1)$**:** {The value of p_i is known up to an additive error of $\sqrt{p_i/2^j}$}.
   1. In w passes, sample $O(2^j p_i \log n + \log n)$ random walks of length w for all nodes in T. This are sufficient to simulate the Step 1 of $2^j O(\log n)$ parallel executions of SINGLERANDOMWALK.
   2. Run $K_{j+1} = 2^j O(\log n)$ independent instances of SINGLERANDOMWALK using the w- length random walks sampled in the previous step. Notice that we have sufficient number of w-length walks from each node, w.h.p.
   3. Estimate the $p_i = n_i/K_{j+1}$ where n_i is the number of walks that use i as a center. This estimate is accurate up to an additive error of $\sqrt{\frac{p}{2^j}}$ *whp* (by Chernoff Bound), for a node with probability p in the actual probability distribution.

We are now ready to prove Theorem 3.1.

PROOF OF THEOREM 3.1. *Correctness:* No w length walk or sampled edge, is ever reused throughout the execution. Also, the number of w length walks we store from the sampled nodes in step 1 of each phase $j+1$ in MULTIPLERANDOMWALK is *sufficient* for K_{j+1} executions of SINGLERANDOMWALK. That is, any time one of K_{j+1} walks hits a sampled node for the first time, there is an unused w length walk for that node, to extend it. This follows from the accuracy of our estimate for p_i obtained from the previous phase j. Each p_i is accurate up to an additive error of $\sqrt{p_i/2^j}$ (this follows from Chernoff bounds since p_i is estimated by using $K_j = 2^{j-1}O(\log n)$ trials). The number of walks in phase $j+1$ that use this node as a center is w.h.p. at most $O(K_{j+1}(p_i + \sqrt{p_i/2^j}) + \log n) \leq O(2^j p_j \log n + \log n)$. This is exactly the number of w-length walks we sample in phase $j+1$..

Space: The main idea why MULTIPLERANDOMWALK does not require too much space is because we only sample as many w-length walks from each center as required in the next phase.

Suppose we do t phases such that $2^t = K$. Total space required would include $O(n\alpha)$ space to store the sampled nodes; to store the w length walks in phase t, we need $O(\sum_{i \in T}(2^t p_i \log n + \log n)) = \tilde{O}(K(\frac{l}{w}) + n\alpha)$ space; and finally, the space for sampling s edges in each execution of HANDLESTUCKNODE amounts to $\tilde{O}(K\frac{l}{w}s + K\frac{1}{\alpha}s)$. Therefore, the algorithm MULTIPLERANDOMWALK uses a total space of $\tilde{O}(K\frac{l}{w}s + K\frac{1}{\alpha}s + \alpha n)$.

Passes: The number of passes required in algorithm MULTIPLERANDOMWALK in any given phase is the same as in SINGLERANDOMWALK, no matter how many walks are being run in that phase. So the total passes required for a given phase is $O(w + \frac{l}{s} + \frac{l}{w\alpha})$. The number of phases run is $\log K$. It follows that the total number of passes in MULTIPLERANDOMWALK is $\tilde{O}(w + \frac{l}{s} + \frac{l}{w\alpha})$. Setting $w = \sqrt{\frac{l}{\alpha}}$ and $s = \sqrt{l\alpha}$, the theorem follows. □

We now show how MULTIPLERANDOMWALK algorithm can be used for estimating the mixing time.

4. APPROXIMATING MIXING TIME

We now present an algorithm to estimate the mixing time of a graph. However, instead of computing the exact mixing time, we compute the time required for *approximate* mixing of a random walk. That is, we compute a length l such that running a random walk for l steps from an initial distribution ends at a node with a probability distribution that is *close* to the stationary distribution. The following definition makes this precise for undirected graphs.

DEFINITION 4.1. *We say that l is the* ϵ-near mixing time *of an undirected graph if the L_1-distance between the steady state distribution and the distribution obtained after a random walk of length l is at most ϵ. Further, l must be the shortest such length that satisfies this condition.*

For directed graphs, we have a weaker definition of *ϵ-near mixing time.*

DEFINITION 4.2. *We say that l is the* ϵ-near mixing time *of a directed graph if the L_1-distance between the distribution obtained after a random walk of length l from any initial distribution, and the distribution obtained after a random walk of length $l + poly(1/\epsilon)$, is at most ϵ.*

REMARK 4.3. *The ϵ-near mixing time satisfies monotonicity property, i.e., if a walk of length l is ϵ-near mixing, so is a walk of length greater than l. The monotonicity property follows from the fact that $||Ax||_1 \leq ||x||_1$ for any transition probability matrix A. This implies that if l is ϵ-near mixing, then $||A^l u - \pi||_1 \leq \epsilon$. By multiplying with A, we have $||A^{l+1}u - \pi||_1 \leq \epsilon$, as $A\pi = \pi$, since π is stationary.*

In this section, specifically, we show the following result.

THEOREM 4.4. *One can find the ϵ-near mixing time for undirected graphs in $\tilde{O}(n\alpha + poly(\epsilon^{-1})(\sqrt{\frac{Mn}{\alpha}} + M\alpha\sqrt{n}))$ space and $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes over the stream, where M is the actual mixing time of the graph. For directed graphs, the space required is $\tilde{O}(n\alpha + poly(\epsilon^{-1})(n^{2/3}\sqrt{\frac{M}{\alpha}} + M\alpha n^{2/3}))$.*

The naive approach to compute the mixing time requires $O(n)$ space and $O(M)$ passes over the input stream, where M is the mixing time. This computes $A^M v$ exactly where v is the initial vector of size n and A the matrix representation

of the graph. It takes n space to maintain this vector, and M passes to multiply by A once in every pass.

The main idea in estimating the mixing time is to run many random walks of length l using the approach described in the previous section, and use these to compute the distribution after l-length random walk. We then compare the distribution at different l, with the stationary distribution, to check if the two distributions are ϵ-near. We need to address the following issues. First, we do not know what value(s) of l to try. Second, we need to compare these distributions with the steady state distribution; while the steady state distribution is easy to compute for an undirected graph, it is hard to compute for directed graphs.

To compare two distributions, we use the technique of Batu et. al. [5] to determine if the distributions are ϵ-near. Their result is summarized in the following theorem.

THEOREM 4.5 ([5]). *Given $\tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ samples of a distribution X over $[n]$, one can test if X is ϵ-near in the L_1 norm to a specific distribution Y.*

We are now ready to prove Theorem 4.4.

PROOF OF THEOREM 4.4. For undirected graphs, the stationary distribution of the random walk is well-known to be $\frac{deg(i)}{2m}$ for node i with degree $deg(i)$, where m is the number of edges in the graph. 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 achived by running MULTIPLERANDOMWALK to obtain $K = \tilde{O}(n^{1/2}poly(\epsilon^{-1}))$ random walks. 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 of ϵ-near mixing time. Note that we can apply binary search as ϵ-near mixing time is a monotonic property.

The result in [5] also provides an approach to determine if two unknown distributions X and Y over $[n]$ are ϵ-close in L_1 norm; however, this requires $\tilde{O}(n^{2/3}poly(\epsilon^{-1}))$ samples from each distribution.

This completes the proof. □

Theorem 4.4 gives some interesting consequences for specific values of α and M. We state some below. In the extreme cases of $\alpha = 1$ and $\alpha = \frac{1}{M}$, we can calculate the ϵ-near mixing time with either of the trade-offs presented in the following corollary.

COROLLARY 4.6. *One can find the ϵ-near mixing time in either $\tilde{O}(n)+poly(\epsilon^{-1})(M\sqrt{n})$ space and $\tilde{O}(\sqrt{M})$ passes, or $\tilde{O}(\frac{n}{M} + M\sqrt{n}poly(\epsilon^{-1}))$ space and $\tilde{O}(M)$ passes.*

Given a mixing time M, we can compute a square-root approximation to the conductance Φ of the graph as $\Theta(1/M) \leq \Phi \leq \Theta(1/\sqrt{M})$ as shown in [23]. The conductance of a graph G with m edges is defined as $\phi(G) = \min_S \frac{E(S,V(G)\,S).m}{E(S).E(V(G)S)}$ where $E(S, V(G)$
$S)$ is the weight of the edges spanning the cut, and $E(S)$ and $E(V(G)$
$S)$ are the weights of the edges on the two sides of the cut. Assuming that the ϵ-near mixing time is close to the actual mixing time, we get the following corollary.

COROLLARY 4.7. *Assuming the ϵ-near mixing time is close to M, the conductance of undirected graphs can be approximated to a quadratic factor in $\tilde{O}(n\alpha+poly(\epsilon^{-1})(\sqrt{\frac{Mn}{\alpha}}+M\alpha\sqrt{n}))$ space and $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes over the stream, where M is the actual mixing time of the graph.*

5. ESTIMATING DISTRIBUTIONS WITH BETTER ACCURACY

In this section, we present an algorithm that has a better space complexity when the accuracy parameter $\epsilon \leq \sqrt{l}/n$. Specifically, if we require a very high accuracy in calculating the PageRank vector, the algorithm presented in this section can be used.

The main idea is to replace the estimation of p_i's in MULTIPLERANDOMWALKS by a quantity that provides more information. We start with the modification of SINGLERANDOMWALK, where we assume that there are infinitely many w length walks out of each sampled node, that can be accessed as an oracle. We then look at the expected number of w length walks that will be used for i. Subsequently, in MODIFIEDMULTIPLERANDOMWALK, we maintain the appropriate number of w length walks from all sampled nodes, in any given phase, using the previous phase to estimate the required number.

In effect, some space is saved during the walks, as s edges need not be sampled from the set of centers S in any walk. However, some space is lost as estimating the number of w length walks to be stored from each i requires more space than estimating just the probability that a w length walk will be used.

We begin by describing the modification to the SINGLERANDOMWALK. The notation in the algorithm below uses T to denote the source node (sampled from an initial distribution) and the sampled nodes obtained by sampling each node with probability α independently. The table W indexed by a sampled node t and *count* stores the end point of the *count*'th w length walk starting at t as $W[t, count]$, for $count \geq 1$ an integer. Note that this table can be populated in t passes, however the space requirement depends on the maximum $count[t]$ for every t. Right now we assume that the table W is infinite and we can obtain a w length walk for any $count[t]$. Unlike in SINGLERANDOMWALK where we defined the set S to keep track of all nodes in T whose w length walks get used up, in MODIFIEDSINGLERANDOMWALK, we do not need S. The algorithm continues extending the walk using the w length walks implicitly stored in the table W until it finds a stuck node. The module MODIFIEDHANDLESTUCKNODE proceeds by sampling s edges out of R where R is the set of stuck nodes visited in the current invocation. In this case, only non-sampled nodes can be a stuck node.

With the algorithm in place, we now need to define q_i, that we use instead of p_i, in MODIFIEDMULTIPLERANDOMWALK. Assume a given set of sampled nodes T.

DEFINITION 5.1. *For any sampled node i, define q_i to be the expected value of $count[i]$ at the end of the execution of l-length random walk using* MODIFIEDSINGLERANDOMWALK.

Notice that the main difference in MODIFIEDSINGLERANDOMWALK as compared to SINGLERANDOMWALK is that there is no set S to store the *centers* whose w length walk has been used up. Every node in the initial sampled set T has the sufficient number of w length walks. So whenever this walk gets, *stuck*, it is stuck at a *non-sampled node.* In

Algorithm 4 ModifiedSingleRandomWalk(u, l)

1: **Input:** Starting node u, and desired walk length l.
2: **Output:** $\mathcal{L}_u$ the random walk from u of length l.
3: $T \leftarrow$ set of nodes obtained by sampling each node independently with probability α.
4: Let $W[t, count] \leftarrow$ the end point of the $count$-th walk of length w from $t \in T$. Let $count[t]$ denote the next index for w length walk of t we will use. Initialize $count[t] = 1$ for all t.
5: Initialize $\mathcal{L}_u$ to a zero length walk starting at u. Let $x \leftarrow u$, the source node.
6: **while** $|\mathcal{L}_u| < l$ **do**
   1. if $(x \in T)$ extend $\mathcal{L}_u$ by appending $W[x, count[x]]$. $count[x] \leftarrow count[x] + 1$. $x \leftarrow$ new end point of $\mathcal{L}_u$. {we increment $count[x]$ so that the next time the walk ends at x, we will use the next w length walk from x stored in the table W.}
   2. if $(x \notin T)$ HandleStuckNode(x, T, $\mathcal{L}_u$, l). {this means that x was not in the initial set of sampled nodes}
7: **end while**

Algorithm 5 ModifiedHandleStuckNode(x, T, $\mathcal{L}_u$, l)

1: $R \leftarrow x$.
2: **while** $|\mathcal{L}_u| < l$ **do**
3: $E \leftarrow$ sample s edges (with repetition) out of each node in R.
4: Extend $\mathcal{L}_u$ as far as possible by walking along the sampled edges in E (on visiting a node in R for the k-th time, use the k-th edge of the s sampled edges from that node).
5: $x \leftarrow$ new end point of $\mathcal{L}_u$ after the extenstion. One of the following cases arise.
   1. if $(x \in R)$ **continue** {no new node is seen}
   2. if $(x \in T)$ **return** {this means that x is a sampled initially; therefore, we can use the next w-length walk from x by accessing the table W}
   3. if $(x \notin T$ and $x \notin R)$ $R \leftarrow R \cup \{x\}$. {this means that x is a new node that has not been visited in this invocation, and x was not in the initial set sampled nodes T}
6: **end while**

this case, s edges are sampled out of all the new nodes visited since the last time a w length walk was used; as before this set is denoted by R.

As we shall see in ModifiedMultipleRandomWalk, extra space goes in estimating the q_i values and ensuring that we have sufficient number of w length walks stored from each sampled node for the subsequent phase. Since we have extra information in q_i as compared to p_i, space is saved in sampling s edges only out of R, instead of $S \cup R$. Hence we get a trade-off between the algorithm presented in this section and that in Section 3.

Estimating q_i using SingleRandomWalk - We now show how to estimate q_i. In the first phase, we use SingleRandomWalk to perform $O(\log n)$ walks. To estimate the q_i's, the entire walks of length l need to be reconstructed. This can be done using the pseudo-random coin tosses as described in Remark 2.4 which adds $\tilde{O}(l)$ to the space requirements. Once the walks of length l are reconstructed, for any walk, start walking from the source node; each time a sampled node (say t) is seen, increment $count[t]$ and skip w steps, and continue walking, till the end. Set q_i to be the average of $count[i]$ over the $O(\log n)$, by this estimation, the average taken over the $O(\log n)$ walks. In phase $j+1$, we run $K_{j+1} = 2^j O(\log n)$ walks and use this to obtain an improved estimate q_i for the next phase.

Notice that after phase $j+1$, we can estimate q_i up to accuracy $q_i \pm \sqrt{(\frac{q_i}{2^{j+1}})\frac{l}{w}}$. This can be seen by dividing q_i by l/w which puts them in the range $[0, 1]$ and then applying Chernoff Bounds.By definition of q_i, $\sum q_i \leq \frac{l}{w}$.

The main theorem is as follows.

Theorem 5.2. ModifiedMultipleRandomWalk *can be used to perform K random walks of length l in $\tilde{O}(\sqrt{(l/\alpha)})$ passes and $\tilde{O}(\alpha n\sqrt{l\alpha} + K\sqrt{\frac{l}{\alpha}} + l)$ space.*

Comparing this with Theorem 3.1, we see that in the space requirement, the term $Kl\alpha$ is no longer there; however, the space of $O(n\alpha)$ in Theorem 3.1 has increased to $O(\alpha n\sqrt{l\alpha})$; additionally, for the first phase where we needed to reconstruct the walks, we incurred an additional space requirement of $O(l)$. Depending on the values and bounds required, one of these theorems is a better than the other. This gives the following corollary similar to Corollary 3.2.

Corollary 5.3. *For any node with probability p in the probability distribution after a walk of length l, one can approximate its probability up to an accuracy of $p \pm \sqrt{p\epsilon}$ for any $\epsilon > 0$ in $\tilde{O}(\sqrt{l/\alpha})$ passes and $\tilde{O}(\alpha n\sqrt{l\alpha} + \frac{1}{\epsilon}\sqrt{\frac{l}{\alpha}} + l)$ space. This also implies a $(1 \pm \sqrt{\frac{\epsilon}{p}})$-approximation ratio for any node with value p in the probability distribution.*

Notice that this is a constant (close to 1) factor approximation for any node with $p \gg \epsilon$.

Proof. *Correctness:* The first point to observe is that the number of w length walks we store from various sampled nodes in each phase in ModifiedMultipleRandomWalk is *sufficient* for the K_j executions of ModifiedSingleRandomWalk. That is, any time one of K_j walks hits a sampled node for the first time, there is an unused w length walk for that node, to extend it. This follows from the accuracy of our estimate for q_i from the previous phase; since each q_i is accurate up to an additive error of $\sqrt{\frac{q_i}{2^j}\frac{l}{w}}$ (this follows from Chernoff bounds since q_i is estimated by using $2^j O(\log n)$ trials), the number of walks in phase $j+1$ that use this node as a center is w.h.p. at most $O(K_{j+1}(q_i + \sqrt{\frac{q_i}{2^j}\frac{l}{w}}) + \frac{l}{w}\log n)$ which is at most $\tilde{O}(2^j q_j + \sqrt{2^j q_j \frac{l}{w}} + \frac{l}{w}) = \tilde{O}(2^j q_j + \frac{l}{w})$. This is exactly the number of w-lenth walks we sample in each phase of the algorithm.
Space: Suppose we do t phases such that $2^t = K$. Total space required would include $O(n\alpha)$ space to store the sampled nodes; to store the w length walks in the phase $(t-1)$, we need $\tilde{O}(\sum_{i \in T}(2^t q_i + \frac{l}{w})) = \tilde{O}(K\sum_{i \in T} q_i + n\alpha\frac{l}{w}) = \tilde{O}(K(\frac{l}{w}) + n\alpha\frac{l}{w})$ space; and finally, the space for sampling s edges in each execution of ModifiedHandleStuckNode

Algorithm 6 ModifiedMultipleRandomWalk($\mathcal{I}$, l, K)

1: **Input**: Distribution of the source nodes, $\mathcal{I}$, and length of the walk, l
2: $T \leftarrow$ set of nodes obtained by sampling each node independently with probability α.
3: Perform phases 1 through $\log K$ as follows -
4: **Phase 1**: Perform $O(\log n)$ walks of length l using SingleRandomWalk. Using Chernoff bound and technique described above for reconstructing the walks and estimating $count[t]$ for all sampled t, we get an estimate all q_i up to an accuracy of $q_i \pm \sqrt{q_i \cdot l/w}$.
5: **Phase** $(j+1)$**:** {spawn K_{j+1} walks and estimate q_i up to accuracy $q_i \pm \sqrt{(\frac{q_i}{2^j})(\frac{l}{w})}$}

- In w passes, sample $K_{j+1} = \tilde{O}(2^j q_i + \frac{l}{w})$ walks from all i. These are sufficient to simulate $2^j O(\log n)$ executions of ModifiedSingleRandomWalk(these walks serve the purpose of the walks stored in the table W).
- Perform $2^j O(\log n)$ random walks using ModifiedSingleRandomWalk and again re-estimate q_i's obtaining a better accuracy. This estimate is obtained by taking the average value of $count[i]$ over the K_{j+1} executions. Notice that, w.h.p., we stored sufficient number of walks for each sampled node. Hence, after the first phase, we never need to reconstruct the walks.

amounts to $\tilde{O}(K\frac{1}{\alpha}s)$. Observe that in ModifiedHandledStuckNode, we only sample s edges out of R instead of $S \cup R$ as in HandleStuckNode. Also, as shown before, $|R| \leq \tilde{O}(1/\alpha)$ since each new node is likely to be a sampled node with probability α. So, the space required for the executions of ModifiedhandleStuckNode is $\tilde{O}(K\frac{1}{\alpha}s)$. Also the first phase required $\tilde{O}(l)$ space to reconstruct the walks obtained from SingleRandomWalk. Therefore, the total space required for this algorithm is $\tilde{O}(K\frac{l}{w}s + K\frac{1}{\alpha}s + \alpha n\frac{l}{w} + l)$.

Passes: The number of passes required is same as in MultipleRandomWalk. So the total passes required for this walk is still $\tilde{O}(w + \frac{l}{s} + \frac{l}{w\alpha})$.

Setting $s = \sqrt{l\alpha}$ and $w = \sqrt{l}\alpha$ gives completes the proof. □

Setting the threshold ϵ to $1/n$ gives the following corollary.

Corollary 5.4. *One can estimate every node's probability distribution after a random walk of length l up to an additive error of $\sqrt{\frac{p}{n}}$ for a node with value p in the probability distribution, in $\tilde{O}(n\sqrt{l})$ space and $\tilde{O}(\sqrt{l})$ passes.*

6. CONCLUSIONS

We presented the following results for graphs presented as edge streams:

1. Algorithm SingleRandomWalk to perform a random walk of length l in $O(\sqrt{l/\alpha})$ passes and $O(n\alpha + \sqrt{l/\alpha})$ space.

2. Algorithm MultipleRandomWalk and algorithm ModifiedMultipleRandomWalk can perform K random walks of length l in $\tilde{O}(\sqrt{l/\alpha})$ passes and $\tilde{O}(n\alpha + K\sqrt{l/\alpha} + Kl\alpha)$ space or $\tilde{O}(n\alpha\sqrt{l\alpha} + K(\sqrt{l/\alpha}))$ space respectively. These algorithms also provide an approach to approximating the probability distribution after a random walk of length l. It follows that every node with probability p in the probability distribution after a random walk of length l can be approximated to an additive error of $\sqrt{p\epsilon}$ using $\tilde{O}(\sqrt{l/\alpha})$ passes and $\min\{\tilde{O}(n\alpha + \frac{1}{\epsilon}\sqrt{l/\alpha} + \frac{1}{\epsilon}l\alpha), \tilde{O}(n\alpha\sqrt{l\alpha} + (1/\epsilon)(\sqrt{l/\alpha}))\}$ space. In particular, the latter algorithm performs better for thresholds $\epsilon \leq \sqrt{l}/n$.

3. We use this technique and present an approach to determine the ϵ-near mixing time, in $\tilde{O}(\sqrt{\frac{M}{\alpha}})$ passes and $\tilde{O}(n\alpha + poly(\epsilon^{-1})(\sqrt{\frac{Mn}{\alpha}} + M\alpha\sqrt{n}))$ space on undirected graphs. The space required to determine the ϵ-near mixing time on directed graphs is $\tilde{O}(n\alpha + poly(\epsilon^{-1})(n^{2/3}\sqrt{\frac{M}{\alpha}} + M\alpha n^{2/3}))$.

Some open questions that arises are:

1. Can we estimate the distribution of nodes with accuracy $\epsilon = 1/n$ using $O(n)$ space?

2. Can one prove any space-pass trade-off bounds? The trivial algorithm to calculate the exact distribution after a random walk of length l requires $O(nl)$ in the space×passes product. Our result stated in Corollary 5.4, for a threshold of $\epsilon = 1/n$, also has the same space-pass trade-off.

3. Are there any lower bounds for space or passes? In particular, are there any strong results for constant passes over the stream?

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments.

7. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[2] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. Local graph partitioning using pagerank vectors. In *Proc. of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 475–486, 2006.

[3] Roy Armoni, Amnon Ta-Shma, Avi Wigderson, and Shiyu Zhou. $sl \leq l^{4/3}$. In *ACM Symposium on Theory of Computing*, pages 230–239, 1997.

[4] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *In Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 2002.

[5] T. Batu, E. Fischer, L. Fortnow, R. Kumar, R. Rubenfeld, and P. White. Testing random variables for independence and identity. In *Proc. of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 442–451, 2001.

[6] L. Bhuvanagiri and S. Ganguly. Estimating entropy over data streams. In *European Symposium on Algorithms (ESA)*, pages 148–159, 2006.

[7] L. Bhuvanagiri, S. Ganguly, D. Kesh, and C. Saha. Simpler algorithm for estimating frequency moments of data streams. In *Proc of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 708–713, 2006.

[8] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. 7th international conference on World Wide Web*, pages 107–117, 1998.

[9] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohle. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.

[10] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *In ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pages 271–282, 2005.

[11] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading of space for passes in graph streaming problems. In *In ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 714–723, 2006.

[12] Uriel Feige. A spectrum of time-space trade-offs for undirected s-t connectivity. *Journal of Computer and System Sciences*, 54(2):305–316, 1997.

[13] J. Feigenbaum, S. Kannan, A McGregor, S Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *In ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 745–754, 2005.

[14] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On the complexity of processing massive, unordered, distributed data. In *CoRR abs/cs/0611108*, 2006.

[15] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *In ACM International Conference on Management of Data, SIGMOD*, pages 58–66, 2001.

[16] S. Guha and A. McGregor. Approximate quantiles and the order of the stream. In *In ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pages 273–279, 2006.

[17] Sudipto Guha and Andrew McGregor. Space-efficient sampling. In *In AISTATS*, pages 169–176, 2007.

[18] Sudipto Guha, Andrew McGregor, and Suresh Venkatasubramanian. Streaming and sublinear approximation of entropy and information distances. In *In ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 733–742, 2006.

[19] Sudipto Guha and Andrew McGrgor. Lower bounds for quantile estimation in random-order and multi-pass streaming. In *ICALP*, pages 704–715, 2007.

[20] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *In External Memory Algorithms, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 50, pages 107–118, 1999.

[21] P. Indyk and D. P. Woodruff. Optimal approximations of the frequency moments of data streams. In *IEEE Symposium on Foundations of Computer Science, FOCS*, pages 283–292, 2003.

[22] Piotr Indyk. Algorithms for dynamic geometric problems over data streams. In *ACM Symposium on Theory of Computing, STOC*, pages 373–380, 2004.

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

[24] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *In COCOON*, pages 710–716, 2005.

[25] G.S. Manku, S. Rajagopalan, and B.G. Lindsay. Randomized sampling techniques for space efficient online computation of order statistics of large datasets. In *In ACM SIGMOD International Conference on Management of Data*, pages 251–262, 1999.

[26] A. McGregor. Finding graph matchings in data streams. In *In APPROX-RANDOM*, pages 170–181, 2005.

[27] Frank McSherry. A uniform approach to accelerated pagerank computation. In *WWW*, pages 575–582, 2005.

[28] T. Sarlos, A. Benczur, K. Csalogany, D. Fogaras, and B. Racz. To randomize or not to randomize: Space optimal summaries for hyperlink analysis. In *In the 15th International World Wide Web Conference, WWW*, pages 297–306, 2006.

[29] John Wicks and Amy R. Greenwald. Parallelizing the computation of pagerank. In *Proc. 5th Workshop On Algorithms And Models For The Web-Graph (WAW)*, pages 202–208, 2007.

[30] David P. Woodruff. Optimal space lower bounds for all frequency moments'. In *In ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 167–175, 2004.