FrogWild! – Fast PageRank Approximations on Graph Engines

Ioannis Mitliagkas
ECE, UT Austin
ioannis@utexas.edu

Michael Borokhovich
ECE, UT Austin
michaelbor@utexas.edu

Alexandros G. Dimakis
ECE, UT Austin
dimakis@austin.utexas.edu

Constantine Caramanis
ECE, UT Austin
constantine@utexas.edu

ABSTRACT

We propose FrogWild, a novel algorithm for fast approximation of high PageRank vertices, geared towards reducing network costs of running traditional PageRank algorithms. Our algorithm can be seen as a quantized version of power iteration that performs multiple parallel random walks over a directed graph. One important innovation is that we introduce a modification to the GraphLab framework that only partially synchronizes mirror vertices. This partial synchronization vastly reduces the network traffic generated by traditional PageRank algorithms, thus greatly reducing the per-iteration cost of PageRank. On the other hand, this partial synchronization also creates dependencies between the random walks used to estimate PageRank. Our main theoretical innovation is the analysis of the correlations introduced by this partial synchronization process and a bound establishing that our approximation is close to the true PageRank vector.

We implement our algorithm in GraphLab and compare it against the default PageRank implementation. We show that our algorithm is very fast, performing each iteration in less than one second on the Twitter graph and can be up to 7× faster compared to the standard GraphLab PageRank implementation.

1. INTRODUCTION

Large-scale graph processing is becoming increasingly important for the analysis of data from social networks, web pages, bioinformatics and recommendation systems. Graph algorithms are difficult to implement in distributed computation frameworks like Hadoop MapReduce and Spark. For this reason several in-memory graph engines like Pregel, Giraph, GraphLab and GraphX [22, 21, 33, 30] are being developed. There is no full consensus on the fundamental abstractions of graph processing frameworks but certain patterns such as vertex programming and the Bulk Synchronous Parallel (BSP) framework seem to be increasingly popular.

PageRank computation [26], which gives an estimate of the importance of each vertex in the graph, is a core component of many search routines; more generally, it represents, de facto, one of the canonical tasks performed using such graph processing frameworks. Indeed, while important in its own right, it also represents the memory, computation and communication challenges to be overcome in large scale iterative graph algorithms.

In this paper we propose a novel algorithm for fast *approximate calculation* of high PageRank vertices. Note that even though most previous works calculate the complete PageRank vector (of length in the millions or billions), in many graph analytics scenarios a user wants a quick estimation of the most important or relevant nodes – distinguishing the 10^{th} most relevant node from the $1,000^{th}$ most relevant is important; the $1,000,000^{th}$ from the $1,001,000^{th}$ much less so. A simple solution is to run the standard PageRank algorithm for fewer iterations (or with an increased tolerance). While certainly incurring less overall cost, the per-iteration cost remains the same; more generally, the question remains whether there is a more efficient way to approximately recover the heaviest PageRank vertices.

There are many real life applications that may benefit from a fast top-k PageRank algorithm. One example is *growing loyalty of influential customers* [1]. In this application, a telecom company identifies the top-k influential customers using the top-k PageRank on the customers' activity (e.g., calls) graph. Then, the company invests its limited budget on improving user experience for these top-k customers, since they are most important for building good reputation. Another interesting example is finding keywords and key sentences in a given text. In [23], the authors show that PageRank performs better than known machine learning techniques for keyword extraction. Each unique word (noun, verb or an adjective) is regarded as a vertex and there is an edge between two words if they occur in close proximity in the text. Using approximate top-k PageRank, we can identify the top-k keywords much faster than obtaining the full ranking. When keyword extraction is used by time sensitive applications or for an ongoing analysis of a large number of documents, speed becomes a crucial factor. The last example we describe here is the application of PageRank for online social networks (OSN). It is important in the context of OSNs to be able to predict which users will

Proceedings of the VLDB Endowment, Vol. 8, No. 8

remain active in the network for a long time. Such *key users* play a decisive role in developing effective advertising strategies and sophisticated customer loyalty programs, both vital for generating revenue [17]. Moreover, the remaining users can be leveraged, for instance for targeted marketing or premium services. It is shown in [17] that PageRank is a much more efficient predictive measure than other centrality measures. The main innovation of [17] is the usage of a mixture of connectivity and activity graphs for PageRank calculation. Since these graphs are highly dynamic (especially the user activity graph), PageRank should be recalculated constantly. Moreover, the *key users* constitute only a small fraction of the total number of users, thus, a fast approximation for the top-PageRank nodes constitutes a desirable alternative to the exact solution.

In this paper we address this problem. Our algorithm (called FrogWild for reasons that will become subsequently apparent) significantly outperforms the simple reduced iterations heuristic in terms of *running time, network communication* and exhibits better scaling. We note that, naturally, we compare our algorithm and reduced-iteration-PageRank within the same framework: we implemented our algorithm in GraphLab PowerGraph and compare it against the built-in PageRank implementation. A key part of our contribution also involves the proposal of what appears to be simply a technically minor modification within the GraphLab framework, but nevertheless results in significant network-traffic savings, and we believe may nevertheless be of more general interest beyond PageRank computations.

Contributions: We consider the problem of fast and efficient (in the sense of time, computation and communication costs) computation of the high PageRank nodes, using a graph engine. To accomplish this we propose and analyze a new PageRank algorithm specifically designed for the graph engine framework and, significantly, we propose a modification of the standard primitives of the graph engine framework (in particular, GraphLab PowerGraph), that enables significant network savings. We explain in further detail both our objectives, and our key innovations.

Rather than seek to recover the full PageRank vector, we aim for the top k PageRank vertices (where k is considered to be approximately in the order of $10 - 1000$). Given an output of a list of k vertices, we define two natural accuracy metrics that compare the true top-k list with our output. The algorithm we propose, FrogWild, operates by starting a small (sublinear in the number of vertices n) number of random walkers (*frogs*) that jump randomly on the directed graph. The random walk interpretation of PageRank enables the frogs to jump to a completely random vertex (teleport) with some constant probability (set to 0.15 in our experiments, following standard convention). After we allow the frogs to jump for time equal to the mixing time of this non-reversible Markov chain, their positions are sampled from the invariant distribution π which is the PageRank vector. The standard PageRank iteration can be seen as the continuous limit of this process (*i.e.*, the frogs become water), which is equivalent to power iteration for stochastic matrices.

The main *algorithmic contributions* of this paper are comprised of the following three innovations. First, we argue that discrete frogs (a quantized form of power iteration) is significantly better for distributed computation, when one is interested only in the large entries of the vector π. This is because each frog produces an independent sample from π. If some entries of π are substantially larger and we only want to determine those, a small number of independent samples suffices. We make this formal using standard Chernoff bounds (see also [29, 14] for similar arguments). On the contrary, during standard PageRank iterations, vertices pass messages to all their out-neighbors since a non-zero amount of water must be transferred. This tremendously increases the network bandwidth especially when the graph engine is over a cluster with many machines.

One major issue with simulating discrete frogs on a graph engine is teleportations. Graph frameworks partition vertices to physical nodes and restrict communication on the edges of the underlying graph. Global random jumps would create dense messaging patterns that would increase communication. Our second innovation is a way of obtaining an identical sampling behavior without teleportations. We achieve this by initiating the frogs at uniformly random positions and having them perform random walks for a life span that follows a geometric random variable. The geometric probability distribution depends on the teleportation probability and can be calculated explicitly.

Our third innovation involves a simple proposed modification for graph frameworks. Most modern graph engines (like GraphLab PowerGraph [16]) employ vertex-cuts as opposed to edge-cuts. This means that each vertex of the graph is assigned to multiple machines so that graph edges see a local vertex mirror. One copy is assigned to be the master and maintains the master version of the vertex data while remaining replicas are mirrors that maintain local cached read–only copies of the data. Changes to the vertex data are made to the master and then replicated to all mirrors at certain synchronization barriers. This architecture is highly suitable for graphs with high-degree vertices (as most real-world graphs are) but has one limitation when used for a few random walks: imagine that vertex v_1 contains one frog that wants to jump to v_2. If vertex v_1 has very high degree, it is very likely that multiple replicas of that vertex exist, possibly one in each machine in the cluster. In an edge-cut scenario only one message would travel from $v_1 \to v_2$, assuming v_1 and v_2 are located in different physical nodes. However, when vertex-cuts are used, the state of v_1 is updated (i.e., contains no frogs now) and this needs to be communicated to all mirrors. It is therefore possible that *a single random walk can create a number of messages equal to the number of machines in the cluster*.

We modify PowerGraph to expose a scalar parameter p_s per vertex. By default, when the framework is running, in each super-step all masters synchronize their programs and vertex data with their mirrors. Our modification is that for each mirror we flip an independent coin and synchronize with probability p_s. Note that when the master does not synchronize the vertex program with a replica, that replica will not be active during that super-step. Therefore, we can avoid the communication and CPU execution by performing limited synchronization in a randomized way.

FrogWild is therefore executed asynchronously but relies on the Bulk Synchronous execution mode of PowerGraph with the additional simple randomization we explained. The name of our algorithm is inspired by HogWild [28], a lock-free asynchronous stochastic gradient descent algorithm proposed by Niu *et al.*. We note that PowerGraph does support an asynchronous execution mode [16] but we implemented

our algorithm by a small modification of synchronous execution. As discussed in [16], the design of asynchronous graph algorithms is highly nontrivial and involves locking protocols and other complications. Our suggestion is that for the specific problem of simulating multiple random walks on a graph, simply randomizing synchronization can give significant benefits while keeping design simple.

While the parameter p_s clearly has the power to significantly reduce network traffic – and indeed, this is precisely born out by our empirical results – it comes at a cost: the standard analysis of the Power Method iteration no longer applies. The main challenge that arises is the theoretical analysis of the FrogWild algorithm. The model is that each vertex is separated across machines and each connection between two vertex copies is present with probability p_s. A single frog performing a random walk on this new graph defines a new Markov Chain and this can be easily designed to have the same invariant distribution π equal to normalized PageRank. The complication is that the trajectories of frogs are *no longer independent*: if two frogs are in vertex v_1 and (say) only one mirror v_1' synchronizes, both frogs will need to jump through edges connected with that particular mirror. Worse still, this correlation effect increases, the more we seek to improve network traffic by further decreasing p_s. Therefore, it is no longer true that one obtains independent samples from the invariant distribution π. Our theoretical contribution is the development of an analytical bound that shows that these dependent random walks still can be used to obtain $\hat{\pi}$ that is provably close to π with high probability. We rely on a coupling argument combined with an analysis of pairwise intersection probabilities for random walks on graphs. In our convergence analysis we use the *contrast bound* [12] for non-reversible chains.

1.1 Notation

Lowercase letters denote scalars or vectors. Uppercase letters denote matrices. The (i, j) element of a matrix A is A_{ij}. We denote the transpose of a matrix A by A'. For a time-varying vector x, we denote its value at time t by x^t. When not otherwise specified, $\|x\|$ denotes the l_2-norm of vector x. We use Δ^{n-1} for the probability simplex in n dimensions, and and $e_i \in \Delta^{n-1}$ for the indicator vector for item i. For example, $e_1 = [1, 0, ...0]$. For the set of all integers from 1 to n we write $[n]$.

2. PROBLEM AND MAIN RESULTS

We now make precise the intuition and outline given in the introduction. We first define the problem, giving the definition of PageRank, the PageRank vector, and therefore its top elements. We then define the algorithm, and finally state our main analytical results.

2.1 Problem Formulation

Consider a directed graph $G = (V, E)$ with n vertices ($|V| = n$) and let A denote its adjacency matrix. That is, $A_{ij} = 1$ if there is an edge from j to i. Otherwise, the value is 0. Let $d_{\text{out}}(j)$ denote the number of *successors* (out-degree) of vertex j in the graph. We assume that all nodes have at least one successor, $d_{\text{out}}(j) > 0$. Then we can define the transition probability matrix P as follows:

$$P_{ij} = A_{ij}/d_{\text{out}}(j). \quad (1)$$

The matrix is left-stochastic, which means that each of its rows sums to 1. We call $G(V, E)$ the *original graph*, as opposed to the PageRank graph, which includes a weighted edge between any pair of vertices. We now define its transition probability matrix, and the PageRank vector.

Definition 1 (PageRank [26]). *Consider the matrix*

$$Q \triangleq (1 - p_T)P + p_T \frac{1}{n} 1_{n\times n}.$$

where $p_T \in [0, 1]$ is a parameter, most commonly set to 0.15. The PageRank vector $\pi \in \Delta^{n-1}$ is defined as the principal right eigenvector of Q. That is, $\pi \triangleq v_1(Q)$. By the Perron-Frobenius theorem, the corresponding eigenvalue is 1. This implies the fixed-point *characterization of the PageRank vector, $\pi = Q\pi$.*

The PageRank vector assigns high values to *important* nodes. Intuitively, important nodes have many important predecessors (other nodes that point to them). This recursive definition is what makes PageRank robust to manipulation, but also expensive to compute. It can be recovered by exact eigendecomposition of Q, but at real problem scales this is prohibitively expensive. In practice, engineers often use a few iterations of the power method to get a "good-enough" approximation.

The definition of PageRank hinges on the left-stochastic matrix Q, suggesting a connection to Markov chains. Indeed, this connection is well documented and studied [2, 15]. An important property of PageRank from its random walk characterization, is the fact that π is the invariant distribution for a Markov chain with dynamics described by Q. A non-zero p_T, also called the *teleportation probability*, introduces a uniform component to the PageRank vector π. We see in our analysis that this implies ergodicity and faster mixing for the random walk.

2.1.1 Top PageRank Elements

Given the true PageRank vector, π and an estimate given by an approximate PageRank algorithm, we define the estimate's top-k accuracy using one of two metrics.

Definition 2 (Mass Captured). *Given distribution $v \in \Delta^{n-1}$, the true PageRank distribution $\pi \in \Delta^{n-1}$ and an integer $k \geq 0$, we define the* mass captured *by v as follows.*

$$\mu_k(v) \triangleq \pi(\text{argmax}_{|S|=k} v(S))$$

For a set $S \subset [n]$, $v(S) \triangleq \sum_{i\in S} v(i)$ denotes the total mass ascribed to the set by the distribution $v \in \Delta^{n-1}$.

Put simply, the set S^* that gets the most mass according to v out of all sets of size k, is evaluated according to π and that gives us our metric. It is maximized by π itself, i.e. the optimal value is $\mu_k(\pi)$.

The second metric we use is the exact identification probability, *i.e.* the fraction of elements in the output list that are also on the true top-k list. Note that the second metric is limited in that it does not give partial credit for high PageRank vertices that are not on the true top-k list. In our experiments in Section 3, we mostly use the normalized captured mass accuracy metric but also report the exact identification probability for some cases – typically the results are similar.

Our algorithm approximates the heaviest elements of the invariant distribution of a Markov Chain, by simultaneously performing multiple random walks on the graph. The main modification to PowerGraph is the exposure of a parameter, p_s, controlling the probability that a given master node synchronizes with any one of its mirrors. Per step, this leads to a proportional reduction in network traffic. The main contribution of this paper is to show that we get results of comparable or improved accuracy, while maintaining this network traffic advantage. We demonstrate this analytically in Section 2.3 and empirically in Section 3.

2.2 Algorithm

During setup, the graph is partitioned using GraphLab's default ingress algorithm. At this point each one of N frogs is born on a vertex chosen uniformly at random. Each vertex i carries a counter initially set to 0 and denoted by $c(i)$. Scheduled vertices execute the following program.

Incoming frogs from previously executed vertex programs, are collected by the `init()` function. At `apply()` every frog dies with probability $p_T = 0.15$. This, along with a uniform starting position, effectively simulates the 15% uniform component from Definition 1.

A crucial part of our algorithm is the change in synchronization behaviour. The <`sync`> step only synchronizes a p_s fraction of mirrors leading to commensurate gains in network traffic (cf. Section 3). This patch on the GraphLab codebase was only a few lines of code. Section 3 contains more details regarding the implementation.

The `scatter()` phase is only executed for edges e incident to a mirror of i that has been synchronized. Those edges draw a binomial number of frogs to send to their other endpoint. The rest of the edges perform no computation. The frogs sent to vertex j at the last step will be collected at the `init()` step when j executes.

Parameter p_T is the teleportation probability from the random surfer model in [26]. To get PageRank using random walks, one could adjust the transition matrix P as described in Definition 1 to get the matrix Q. Alternatively, the process can be replicated by a random walk following the original matrix P, and teleporting at every time, with probability p_T. The destination for this teleportation is chosen uniformly at random from $[n]$. We are interested in the position of a walk at a predetermined point in time as that would give us a sample from π. This holds as long as we allow enough time for mixing to occur.

Due to the inherent markovianity in this process, one could just consider it starting from the last teleportation before the predetermined stopping time. When the stopping time is late enough, the number of steps performed between the last teleportation and the predetermined stopping time, denoted by X, is geometrically distributed with parameter p_T. This follows from the time-reversibility in the teleportation process: inter-teleportation times are geometrically distributed, so as long as the first teleportation event happens before the stopping time, then $X \sim \text{Geom}(\text{p}_\text{T})$.

This establishes that, the FrogWild! process – where a frog performs a geometrically distributed number of steps following the original transition matrix P – closely mimics a random walk that follows the adjusted transition matrix, Q. In practice, we stop the process after t steps to get a good approximation. To show our main result, Theorem 1, we analyze the latter process.

FrogWild! vertex program

Input parameters: p_s, $p_T = 0.15$, t

apply(i) $K(i) \leftarrow$ [# incoming frogs]

If t steps have been performed:

$c(i) \leftarrow c(i) + K(i)$

HALT

For every incoming frog:

With probability p_T, frog dies:

$c(i) \leftarrow c(i) + 1$,

$K(i) \leftarrow K(i) - 1$.

<**sync**> For every *mirror* m of vertex i:

With probability p_s:

Synchronize state with mirror m.

scatter($e = (i, j)$) [Only on synchronized mirrors]

Generate Binomial number of frogs:

$$x \sim Bin\left(K(i), \frac{1}{d_{\text{out}}(i)p_s}\right)$$

Send x frogs to vertex j: `signal(j,x)`

Using a binomial distribution to independently generate the number of frogs in the `scatter()` phase closely models the effect of random walks. The marginal distributions are correct, and the number of frogs, that did not die during the `apply()` step, is preserved in expectation. For our implementation we resort to a more efficient approach. Assuming $K(i)$ frogs survived the `apply()` step, and M mirrors where picked for synchronization, then we send $\lceil \frac{K(i)}{M} \rceil$ frogs to $\min(K(i), M)$ mirrors. If the number of available frogs is less than the number of synchronized mirrors, we pick $K(i)$ arbitrarily.

2.3 Main Result

Our analytical results essentially provide a high probability guarantee that our algorithm produces a solution that approximates well the PageRank vector. Recall that the main modification of our algorithm involves randomizing the synchronization between master nodes and mirrors. For our analysis, we introduce, in Appendix A, a broad model to deal with partial synchronization.

Our results tell us that partial synchronization does not change the distribution of a single random walk. To our statements clear, we need the simple definition.

Definition 3. *We denote the state of random walk i at its t^{th} step by s_i^t.*

Then, we see that $\mathbb{P}\big(s_1^{t+1} = i \big| s_1^t = j\big) = 1/d_{\text{out}}(j)$, and $x_1^{t+1} = Px_1^t$. This follows simply by the symmetry assumed in Definition 8. Thus if we were to sample serially, the modification of the algorithm controlling (limiting) synchronization would not affect each sample, and hence would not affect our estimate of the invariant distribution. However, we start multiple (all) random walks simultaneously. In this

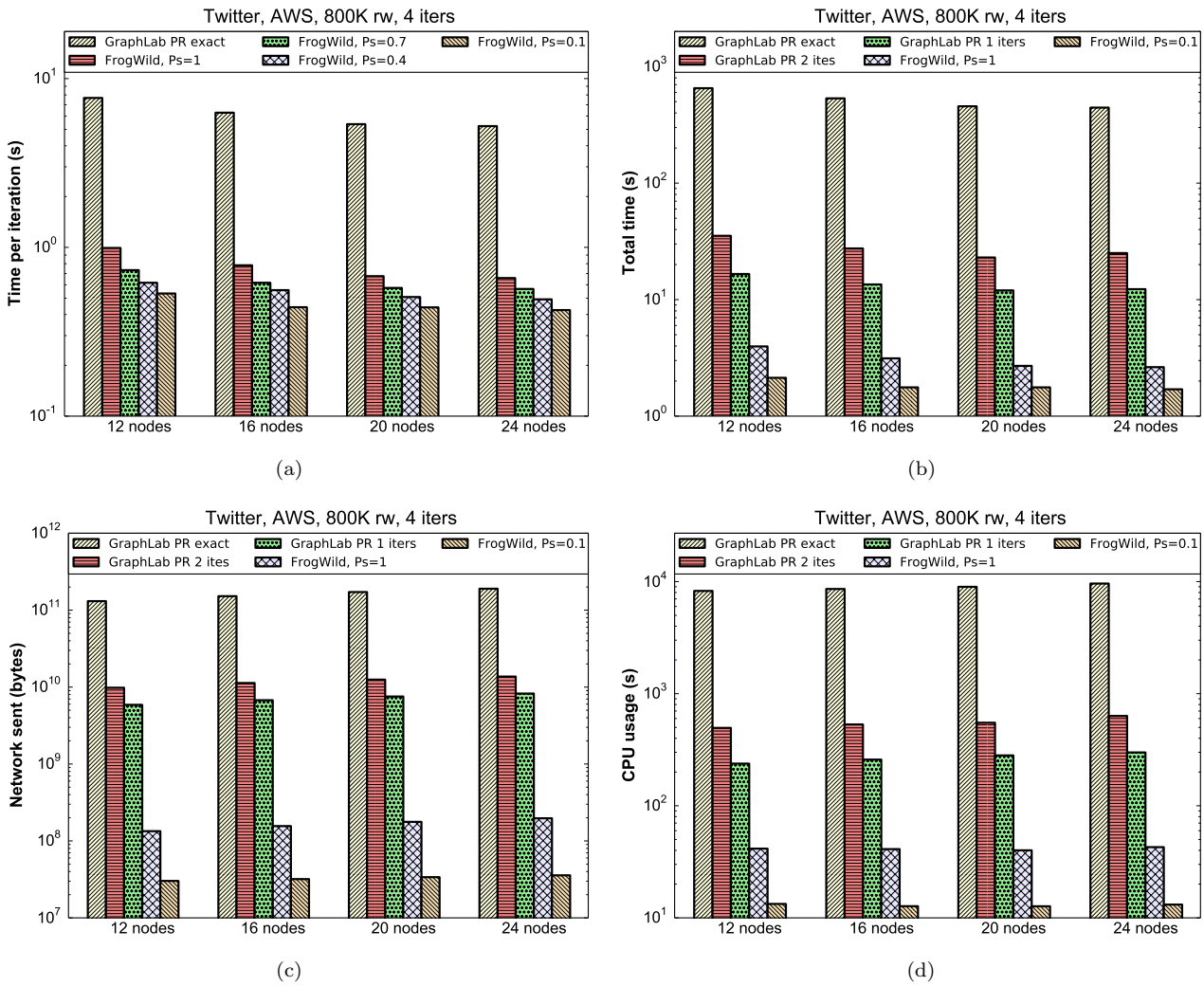


Figure 1: PageRank performance for various number of nodes. Graph: Twitter; system: AWS (Amazon Web Services); FrogWild parameters: 800K initial random walks and 4 iterations. (a) Running time per iteration. (b) Total running time of the algorithms. (c) Total network bytes sent by the algorithm during the execution (does not include ingress time). (d) Total CPU usage time. Notice, this metric may be larger than the total running time since many CPUs run in parallel.

setting, the fundamental analytical challenge stems from the fact that random walks that intersect are correlated. The key to our result is that we can control the effect of this correlation, as a function the parameter p_s and the *pairwise probability* that two random walks intersect. We define this formally.

DEFINITION 4. *Suppose two walkers l_1 and l_2 start at the same time and perform t steps. The probability that they meet is defined as follows.*

$$p_\cap(t) \triangleq \mathbb{P}\left(\exists\ \tau \in [0,t],\ \text{s.t. } s^\tau_{l_1} = s^\tau_{l_2}\right) \tag{2}$$

DEFINITION 5 (ESTIMATOR). *Given the positions of N random walks at stopping time t, $\{s^t_l\}_{l=1}^N$, we define the following estimator for the invariant distribution π.*

$$\hat{\pi}_N(i) \triangleq \frac{\left|\{l : l \in [N], s^t_l = i\}\right|}{N} = \frac{c(i)}{N} \tag{3}$$

Here $c(i)$ refers to the tally maintained by the FROGWILD! *vertex program.*

Now we can state the main result. Here we give a guarantee for the quality of the solution furnished by our algorithm.

THEOREM 1 (MAIN THEOREM). *Consider N frogs following the* FROGWILD! *process (Section 2.2), under the erasure model of Definition 8. The frogs start at independent locations, distributed uniformly and stop after a geometric number of steps or, at most, t steps. The estimator $\hat{\pi}_N$ (Definition 5), captures mass close to the optimal. Specifically, with probability at least $1-\delta$,*

$$\mu_k(\hat{\pi}_N) \geq \mu_k(\pi) - \epsilon,$$

where

$$\epsilon < \sqrt{\frac{(1-p_T)^{t+1}}{p_T}} + \sqrt{\frac{k}{\delta}\left[\frac{1}{N} + (1-p_s^2)p_\cap(t)\right]}. \tag{4}$$

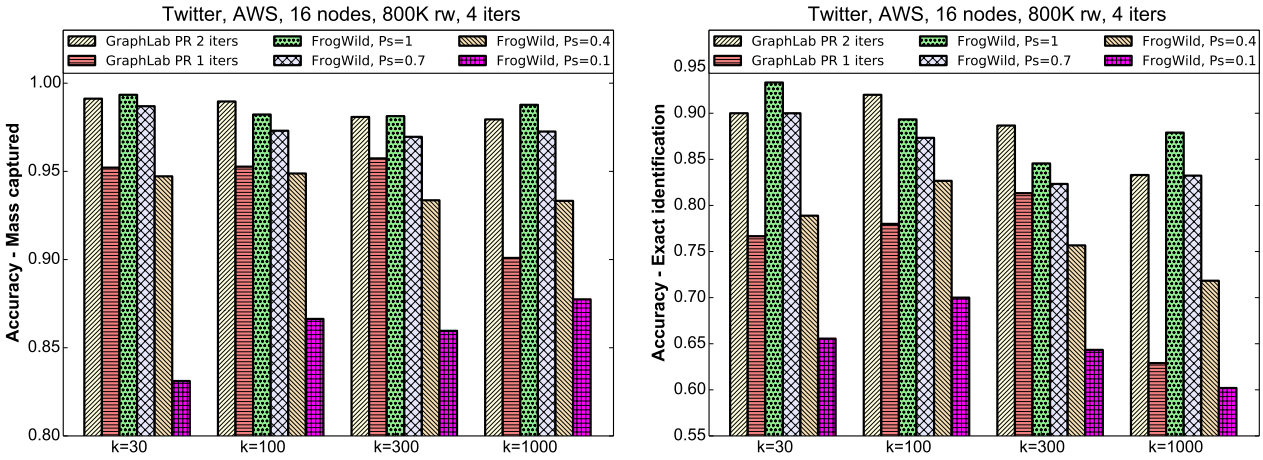


Figure 2: PageRank approximation accuracy for various number of top-k PageRank vertices. Graph: Twitter; system: AWS (Amazon Web Services) with 16 nodes; FrogWild parameters: 800K initial random walks and 4 iterations. (a) Mass captured. The total PageRank that the reported top-k vertices worth in the exact ranking. (b) Exact identification. The number of vertices in the intersection of the reported top-k and the exact top-k lists.

REMARK 6 (SCALING). *The result in Theorem 1 immediately implies the following scaling for the number of iterations and frogs respectively. They both depend on the maximum captured mass possible, $\mu_k(\pi)$ and are sufficient for making the error, ϵ, of the same order as $\mu_k(\pi)$.*

$$t = O\left(\log \frac{1}{\mu_k(\pi)}\right), \qquad N = O\left(\frac{k}{\mu_k(\pi)^2}\right)$$

The proof of Theorem 1 is deferred to Appendix B.1. The guaranteed accuracy via this result also depends on the probability that two walkers will intersect. Via a simple argument, that probability is the same as the meeting probability for independent walks. The next theorem bounds this probability.

THEOREM 2 (INTERSECTION PROBABILITY). *Consider two independent random walks obeying the same ergodic transition probability matrix, Q with invariant distribution π, as described in Definition 1. Furthermore, assume that both of them are initially distributed uniformly over the state space of size n. The probability that they meet within t steps, is bounded as follows,*

$$p_\cap(t) \leq \frac{1}{n} + \frac{t\|\pi\|_\infty}{p_T},$$

where $\|\pi\|_\infty$, denotes the maximal element of the vector π.

The proof is based on the observation that the l_∞ norm of a distribution controls the probability that two independent samples coincide. We show that for all steps of the random walk, that norm is controlled by the l_∞ norm of π. We defer the full proof to Appendix B.2.

A number of studies, give experimental evidence (e.g. [8]) suggesting that PageRank values for the web graph follow a power-law distribution with parameter approximately $\theta = 2.2$. That is true for the tail of the distribution – the largest values, hence of interest to us here – regardless of the choice of p_T. The following proposition bounds the value of the heaviest PageRank value, $\|\pi\|_\infty$.

PROPOSITION 7 (MAXIMUM OF POWER-LAW). *Let $\pi \in \Delta^{n-1}$ follow a power-law distribution with parameter θ and minimum value p_T/n. Its maximum element, $\|\pi\|_\infty$, is at most $n^{-\gamma}$, with probability at least $1 - cn^{\gamma - \frac{1}{\theta-1}}$, for some universal constant c.*

PROOF. The expected maximum value of n independent draws from a power-law distribution with parameter θ, is shown in [25] to be

$$\mathbb{E}x_{max} = O(n^{-\frac{1}{\theta-1}}).$$

Simple application of Markov's inequality, gives us the statement. □

Assuming $\theta = 2.2$ and picking, for example, $\gamma = 0.5$, we get

$$\mathbb{P}(\|\pi\|_\infty > 1/\sqrt{n}) \leq cn^{-1/3}.$$

This implies that with probability at least $1 - cn^{-1/3}$ the meeting probability is bounded as follows.

$$p_\cap(t) \leq \frac{1}{n} + \frac{t}{p_T\sqrt{n}}.$$

One would usually take a number of steps t that are either constant or logarithmic with respect to the graph size n. This implies that for many reasonable choices of set size k and acceptable probability of failure δ, the meeting probability vanishes as n grows. Then we can make the second term of the error in (4) arbitrarily small by controlling the number of frogs, N.

2.4 Prior Work

There is a very large body of work on computing and approximating PageRank on different computation models (*e.g.* see [10, 13, 29, 14, 4] and references therein). To the best of our knowledge, our work is the first to specifically design an approximation algorithm for high-PageRank nodes for graph engines. Another line of work looks for *Personalized PageRank* (PPR) scores. This quantifies the influence

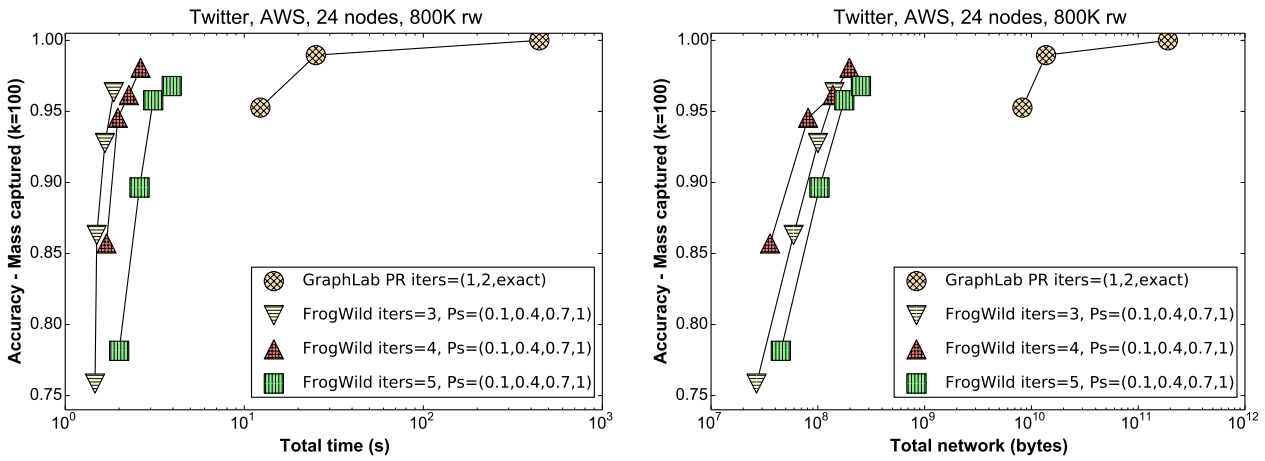


Figure 3: PageRank approximation accuracy with the "Mass captured" metric for top-100 vertices. Graph: Twitter; system: AWS (Amazon Web Services) with 24 nodes; FrogWild parameters: 800K initial random walks. (a) - Accuracy versus total running time. (b) - Accuracy versus total network bytes sent.

an arbitrary node i has on another node j, cf. recent work [20] and discussion therein. In [6], the top-k approximation of PPR is studied. However, PPR is not applicable in our case, as we are looking for an answer close to a *global* optimum.

In [5], a random-walks-based algorithm is proposed. The authors provide some insightful analysis of different variations of the algorithm. They show that starting a single walker from every node, is sufficient to achieve a good global approximation. We focus on capturing a few nodes with a lot of mass, hence we can get away with orderwise much fewer frogs than $O(n)$. This is important for achieving low network traffic when the algorithm is executed on a distributed graph framework. Figure 8 shows linear reduction in network traffic when the number of initial walkers decreases. Furthermore, our method does not require waiting for the last frog to naturally expire (note that the geometric distribution has infinite support). We impose a very short time cut-off, t, and exactly analyze the penalty in captured mass we pay for it in Theorem 1.

One natural question is how our algorithm compares to, or can be complemented by, graph sparsification techniques. One issue here is that graph sparsification crucially depends on the similarity metric used. Well-studied properties that are preserved by sparsification methods include lengths of shortest paths between vertices (such sparsifiers are called Spanners, see e.g. [27]), cuts between subsets of vertices [9] and more generally quadratic forms of the graph Laplacian [31, 7], see [7] and references therein for a recent overview. To the best of our knowledge, there are no known graph sparsification techniques that preserve vertex PageRank.

One natural heuristic that one may consider is to independently flip a coin and delete each edge of the graph with some probability r. Note that this is crucially different from spectral sparsifiers [31, 7] that choose these probabilities using a process that is already more complicated than estimating PageRank. This simple heuristic of independently deleting edges indeed accelerates the estimation process for high-PageRank vertices. We compare FrogWild to this uniform sparsification process in Figure 5. We present here results for 2 iterations of the GraphLab PR on the sparsified graph. Note that running only one iteration is not interesting since it actually estimates only the in-degree of a node which is known in advance (i.e., just after the graph loading) in a graph engine framework. It can be seen in Figure 5 that even when only two iterations are used on the sparsified graph the running time is significantly worse compared to FrogWild and the accuracy is comparable.

Our base-line comparisons come from the graph framework papers since PageRank is a standard benchmark for running-time, network and other computations. Our implementation is on GraphLab (PowerGraph) and significantly outperforms the built-in PageRank algorithm. This algorithm is already shown in [16, 30] to be significantly more efficient compared to other frameworks like Hadoop, Spark, Giraph *etc.*

3. EXPERIMENTS

In this section we compare the performance of our algorithm to the PageRank algorithm shipped with GraphLab v2.2 (PowerGraph) [21]. The fact that GraphLab is the fastest distributed engine for PageRank is established experimentally in [30]. We focus on two algorithms: the basic built-in algorithm provided as part of the GraphLab *graph analytics toolkit*, referred to here as GraphLab PR, and FrogWild. Since we are looking for a top-k approximation and GraphLab PR is meant to find the entire PageRank vector, we only run it for a small number of iterations (usually 2 are sufficient). This gives us a good top-k approximation and is much faster than running the algorithm until convergence. We also fine tune the algorithm's tolerance parameter to get a good but fast approximation.

We compare several performance metrics, namely: running time, network usage, and accuracy. The metrics do not include time and network usage required for loading the graph into GraphLab (known as the *ingress time*). They reflect only the execution stage.

3.1 The Systems

We perform experiments on two systems. The first system is a cluster of 20 virtual machines, created using VirtualBox 4.3 [32] on a single physical server. The server is based on an Intel® Xeon® CPU E5-1620 with 4 cores at 3.6 GHz, and 16 GB of RAM. The second system, comprises of a cluster of up to 24 EC2 machines on AWS (Amazon web services) [3]. We use m3.xlarge instances, based on Intel® Xeon® CPU E5-2670 with 4 vCPUs and 15 GB RAM.

3.2 The Data

For the VirtualBox system, we use the LiveJournal graph [19] with 4.8M vertices and 69M edges. For the AWS system, in addition to the LiveJournal graph, we use the Twitter graph [18] which has 41.6M nodes and 1.4B edges.

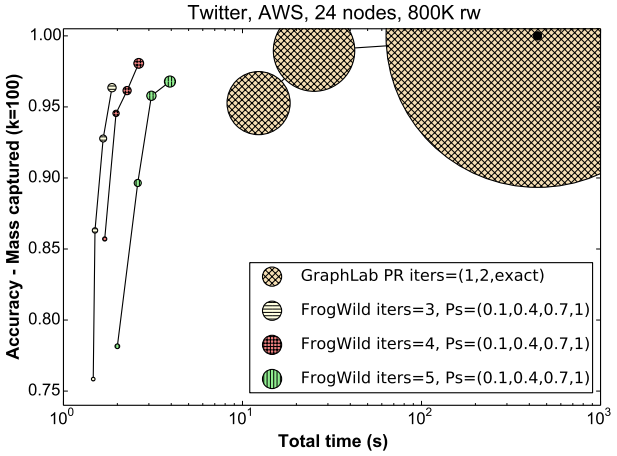


Figure 4: Accuracy versus total running time. Graph: Twitter; system: AWS (Amazon Web Services) with 24 nodes; FrogWild parameters: 800K initial random walks. The area of each circle is proportional to the total network bytes sent by the specific algorithm.

3.3 Implementation

FrogWild is implemented on the standard GAS (gather, apply, scatter) model. We implement `init()`, `apply()`, and `scatter()`. The purpose of `init()` is to collect the random walks sent to the node by its neighbors using `scatter()` in the previous iteration. In the first iteration, `init()` generates a random fraction of the initial total number of walkers. This implies that the initial walker locations are randomly distributed across nodes. FrogWild requires the length of random walks to be geometrically distributed (see Section 2.2). For the sake of efficiency, we impose an upper bound on the length of random walks. The algorithm is executed for the constant number of iterations (experiments show good results with even 3 iterations) after which all the random walks are stopped simultaneously. The `apply()` function is responsible for keeping track of the number of walkers that have stopped on each vertex and `scatter()` distributes the walkers still alive to the neighbors of the vertex. The `scatter()` phase is the most challenging part of the implementation. In order to reduce information exchange between machines, we use a couple of ideas.

First, we notice that random walks do not have identity. Hence, random walks destined for the same neighbor can be combined into a single message. The main optimization and most significant part of our work is modifying the GraphLab engine to support *randomized synchronization*, as described in Section 2.2. We expose the synchronization probability $p_s \in [0, 1]$ to the user as a small extension to the GraphLab API. It describes the fraction of replicas that will be synchronized. Replicas not synchronized remain idle for the upcoming scatter phase. The source changes in the engine are a matter of a few (about 10) lines of code. Using this feature is completely optional; i.e., setting $p_s = 1$ results in using the original engine, leaving other analytic workloads unaffected. However, any random walk or "gossip" style algorithm (that sends a single messages to a random subset of its neighbors) can benefit by reducing p_s. Our modification of the GraphLab engine as well as the FrogWild vertex program can be found in [11].

3.4 Results

FrogWild is significantly faster and uses less network and CPU compared to GraphLab PR. Let us start with the Twitter graph and the AWS system. In Figure 1(a) we see that, while GraphLab PR takes about 7.5 seconds per iteration (for 12 nodes), FrogWild takes less than 1 sec, achieving more than a $7x$ speedup. Reducing the value of p_s decreases the running time. We see a similar picture with the total running time of the algorithms in Figure 1(b)).

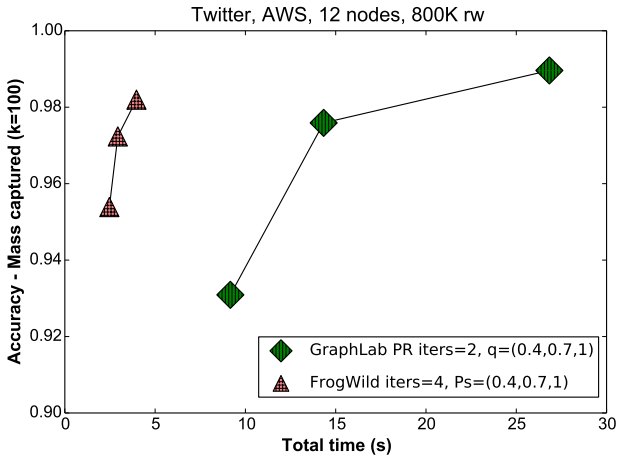


Figure 5: Accuracy versus total running time. Graph: Twitter; system: AWS (Amazon Web Services) with 12 nodes; FrogWild parameters: 800K initial random walks. $q = 1 - r$ is the probability of *keeping* an edge in the sparsification process.

We plot network performance in Figure 1(c). We get a $1000x$ improvement compared to the exact GraphLab PR, and more than $10x$ with respect to doing 1 or 2 iterations of GraphLab PR. In Figure 1(d) we can see that the total CPU usage reported by the engine is also much lower for FrogWild.

We now turn to compare the approximation metrics for the PageRank algorithm. For various k, we check the two accuracy metrics: Mass captured (Figure 2(a)) and the Exact identification (Figure 2(b)). Mass captured – is the total

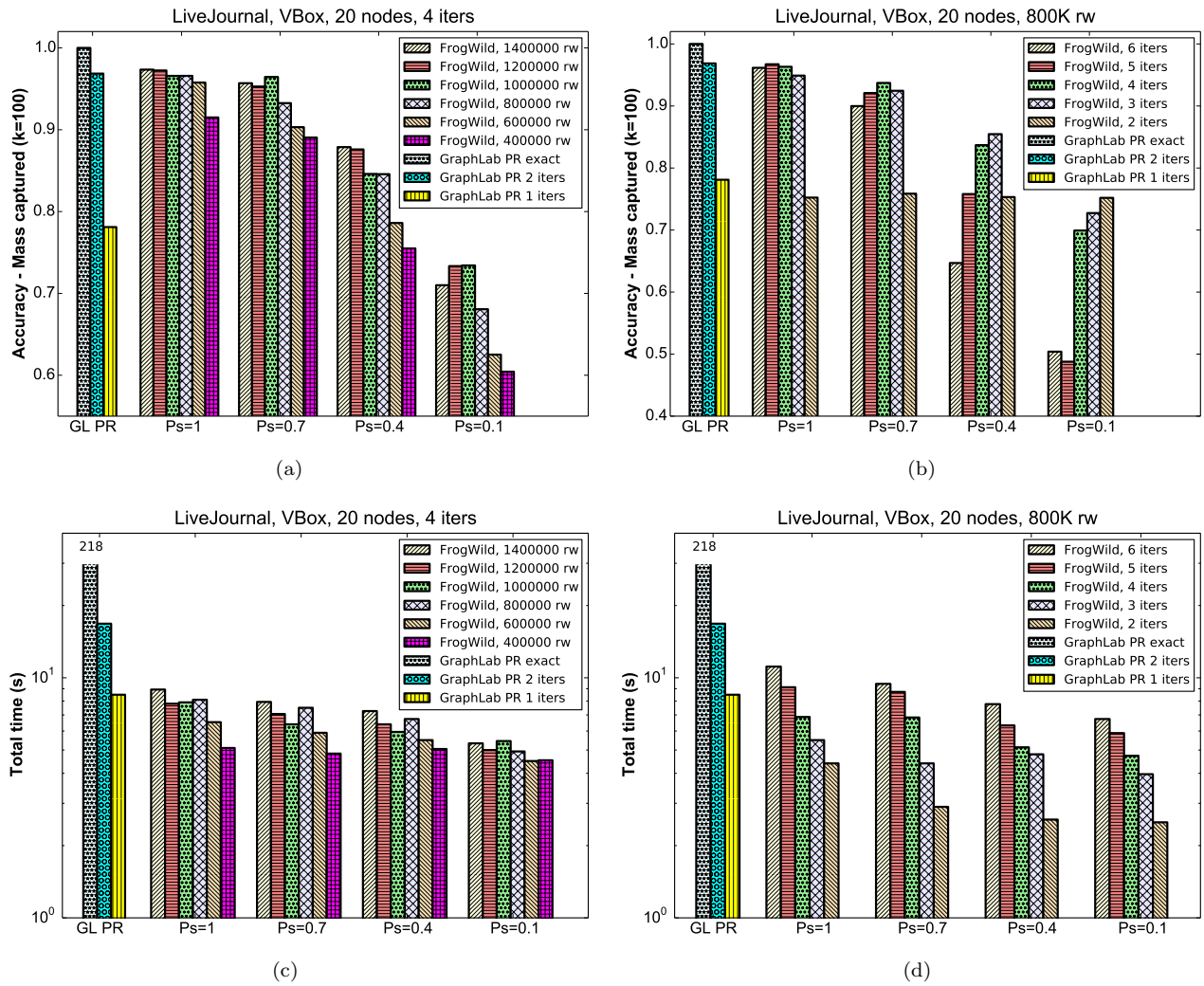


Figure 6: Graph: LiveJournal; system: VirtualBox with 20 nodes. (a) Accuracy for various number of initial random walks in the FrogWild (with 4 iterations). (b) Accuracy for various number of iterations of FrogWild (with 800K initial random walks). (c) Total running time for various number of initial random walks in the FrogWild (with 4 iterations). (d) Total running time for various number of iterations of FrogWild (with 800K initial random walks).

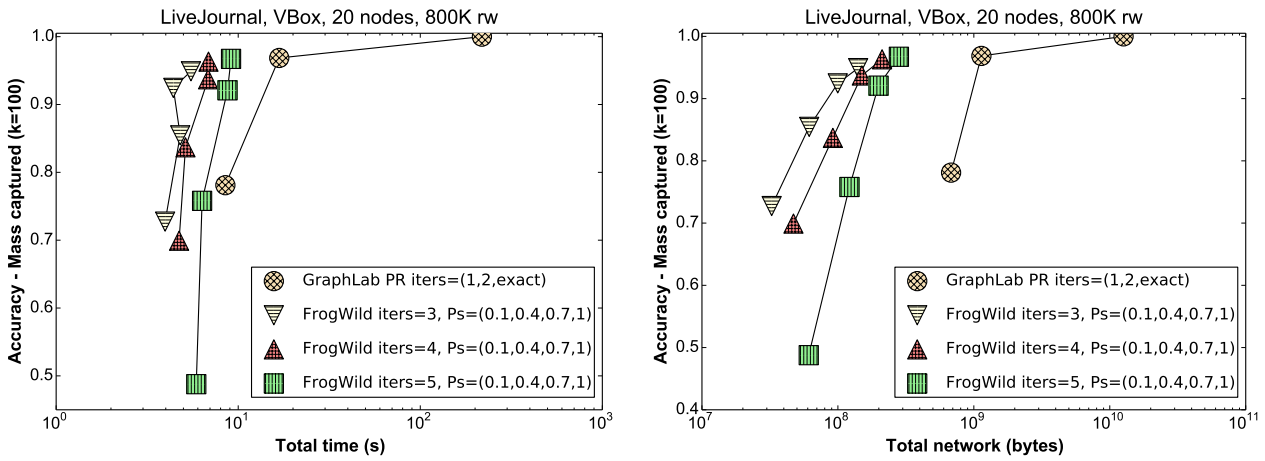
PageRank that the reported top-k vertices worth in the exact ranking. Exact identification – is the number of vertices in the intersection of the reported top-k and the exact top-k lists. We can see that the approximation achieved by the FrogWild for $p_s = 1$ and $p_s = 0.7$ always outperforms the GraphLab PR with 1 iteration. The approximation achieved by the FrogWild with $p_s = 0.4$ is relatively good for the both metrics, and with $p_s = 0.1$ is reasonable for the *Mass captured* metrics.

In Figure 3 we can see the trade-off between the accuracy, total running time, and the network usage. The performance of FrogWild is evaluated for various number of iterations and the values of p_s. The results show that with the accuracy comparable to GraphLab PR, FrogWild has much less running time and network usage. Figure 4 illustrates how much network traffic we save using FrogWild. The area of each circle is proportional to the number of bytes sent by each algorithm.

We also compare FrogWild to an approximation strategy that uses a simple sparsification technique described in Section 2.4. First, the graph is sparsified by deleting each edge with probability r, then GraphLab PR is executed. In Figure 5, we can see that FrogWild outperforms this approach in terms running time while achieving comparable accuracy.

Finally, we plot results for the LiveJournal graph on the VirtualBox system. Figures 6(a,b) show the effect of the number of walkers, N, and the number of iterations for FrogWild on the achieved accuracy. Good accuracy and running time (see Figure 6(c,d)) are achieved for $800K$ initial random walks and 4 iterations of FrogWild. Similar to the Twitter graph, also for the LiveJournal graph we can see, in Figure 7, that our algorithm is faster and uses much less network, while still maintaining good PageRank accuracy. By varying the number of initial random walks and the number of iterations we can fine-tune the FrogWild for the optimal accuracy-speed trade-off.

Interestingly, for both graphs (Twitter and LiveJournal), reasonable parameters are: 800K initial random walks and 4 iterations, despite the order of magnitude difference in



(a) (b)

Figure 7: Graph: LiveJournal; system: VirtualBox with 20 nodes; FrogWild parameters: 800K initial random walks. (a) Accuracy versus total running time. (b) Accuracy versus total network bytes sent.

the graph sizes. This implies slow growth for the necessary number of frogs with respect to the size of the graph. This scaling behavior is tough to check in practice, but it is explained by our analysis. Specifically, Remark 6 shows that the number of frogs should scale as $N = O\left(\frac{k}{\mu_k(\pi)^2}\right)$.

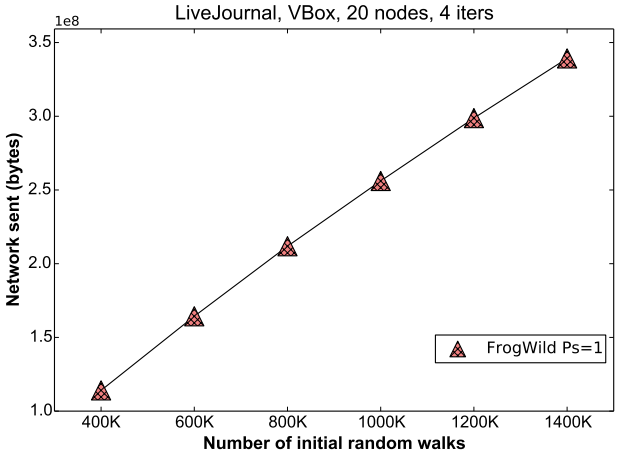


Figure 8: Network usage of FrogWild versus the number of initial random walks. Graph: LiveJournal; system: VirtualBox with 20 nodes; FrogWild parameters: 4 iterations.

4. ACKNOWLEDGMENTS

The authors would like to thank Sanjay Shakkottai and Joe Neeman for their helpful feedback. This research was supported by NSF CCF 1344179, 1344364, 1407278, 1422549, NSF 1056028, NSF 1302435, NSF 1116955, DARPA STTR and an ARO YIP award. This research was also partially supported by the U.S. Dept. of Transportation through the Data-Supported Transportation Operations and Planning (D-STOP) Tier 1 University Transportation Center.

5. REFERENCES

[1] Teradata. `http://www.teradata.com/Resources/Videos/Grow-Loyalty-of-Influential-Customers`. Accessed: 2014-11-30.

[2] A. Agarwal and S. Chakrabarti. Learning random walks to rank nodes in graphs. In *Proceedings of the 24th international conference on Machine learning*, pages 9–16. ACM, 2007.

[3] Amazon web services. http://aws.amazon.com, 2014.

[4] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. S. Mirrokni, and S.-H. Teng. Local computation of pagerank contributions. In *Algorithms and Models for the Web-Graph*, pages 150–165. Springer, 2007.

[5] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM J. Numer. Anal.*, 45(2):890–904, Feb. 2007.

[6] K. Avrachenkov, N. Litvak, D. Nemirovsky, E. Smirnova, and M. Sokol. Monte carlo methods for top-k personalized pagerank lists and name disambiguation. *CoRR*, abs/1008.3775, 2010.

[7] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng. Spectral sparsification of graphs: Theory and algorithms. *Commun. ACM*, 56(8):87–94, Aug. 2013.

[8] L. Becchetti and C. Castillo. The distribution of pagerank follows a power-law only for particular values of the damping factor. In *Proceedings of the 15th international conference on World Wide Web*, pages 941–942. ACM, 2006.

[9] A. A. Benczúr and D. R. Karger. Approximating s-t minimum cuts in Õ(n2) time. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 47–55, New York, NY, USA, 1996. ACM.

[10] P. Berkhin. A survey on pagerank computing. *Internet Mathematics*, 2(1):73–120, 2005.

[11] M. Borokhovich and I. Mitliagkas. FrogWild! code repository. `https://github.com/michaelbor/frogwild`, 2014. Accessed: 2014-10-30.

[12] P. Bremaud. *Markov chains: Gibbs fields, Monte Carlo simulation, and queues*, volume 31. springer, 1999.

[13] A. Z. Broder, R. Lempel, F. Maghoul, and J. Pedersen. Efficient pagerank approximation via graph aggregation. *Information Retrieval*, 9(2):123–138, 2006.

[14] A. Das Sarma, D. Nanongkai, G. Pandurangan, and P. Tetali. Distributed random walks. *Journal of the ACM (JACM)*, 60(1):2, 2013.

[15] S. Fortunato and A. Flammini. Random walks on directed networks: the case of pagerank. *International Journal of Bifurcation and Chaos*, 17(07):2343–2353, 2007.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[17] J. Heidemann, M. Klier, and F. Probst. Identifying key users in online social networks: A pagerank based approach. In *ICIS'10*, 2010.

[18] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[19] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[20] P. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. Fast-ppr: Scaling personalized pagerank estimation for large graphs. *arXiv preprint arXiv:1404.3181*, 2014.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[23] R. Mihalcea and P. Tarau. Textrank: Bringing order into texts. Association for Computational Linguistics, 2004.

[24] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis. FrogWild! – Long Technical Version. `http://mitliagkas.github.io/papers/FrogWild.pdf`, 2015. Accessed: 2015-03-04.

[25] M. E. Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[27] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 77–85, New York, NY, USA, 1987. ACM.

[28] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[29] A. D. Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. *Journal of the ACM (JACM)*, 58(3):13, 2011.

[30] N. Satish, N. Sundaram, M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets.

[31] D. Spielman and N. Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[32] VirtualBox 4.3. www.virtualbox.org, 2014.

[33] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

APPENDIX

A. EDGE ERASURE MODEL

Definition 8 (Edge Erasure Model). *An edge erasure model is a process that is independent from the random walks (up to time t) and temporarily erases a subset of all edges at time t. The event $E^t_{i,j}$ represents the erasure of edge (i,j) from the graph for time t. The edge is not permanently removed from the graph, it is just disabled and considered again in the next step. The edge erasure models we study satisfy the following properties.*

1. *Edges are erased independently for different vertices,*
$$\mathbb{P}(E^t_{i,j}, E^t_{i,k}) = \mathbb{P}(E^t_{i,j})\mathbb{P}(E^t_{i,k})$$
and across time,
$$\mathbb{P}(E^t_{i,j}, E^s_{i,j}) = \mathbb{P}(E^t_{i,j})\mathbb{P}(E^s_{i,j}).$$
2. *Each outgoing edge is preserved (not erased) with probability at least p_s.*
$$\mathbb{P}(\overline{E^t_{i,j}}) \geq p_s$$
3. *Erasures do not exhibit significant negative correlation. Specifically,*
$$\mathbb{P}(\overline{E^t_{i,j}}|\overline{E^t_{i,k}}) \geq p_s.$$
4. *Erasures in a neighbourhood are symmetric. Any subset of out-going edges of vertex i, will be erased with exactly the same probability as another subset of the same cardinality.*

The main two edge erasure models we consider are described here. They both satisfy all required properties. Our theory holds for both[1], but in our implementation and experiments we use "At Least One Out-Edge Per Node."

Example 9 (Independent Erasures). *Every edge is preserved independently with probability p_s.*

Example 10 (At Least One Out-Edge Per Node). *This edge erasure model, decides all erasures for node i independently, like* Independent Erasures*, but if all out-going edges for node i are erased, it draws and enables one of them uniformly at random.*

[1] *Independent Erasures* can lose some walkers, when it temporarily leads to some nodes having zero out-degree.

B. THEOREM PROOFS

B.1 Proof of Theorem 1

In this section we provide a proof sketch for our main results. We start from simple processes and slowly introduce the analytical intricacies of our system one-by-one giving guarantees on the performance of each stage. Detailed proofs for the lemmata are deferred to the long version of this paper [24].

PROCESS 11 (FIXED STEP). *Independent walkers start on nodes selected uniformly at random and perform random walks on the* augmented *graph. This means that teleportation happens with probability p_T and the walk is described by the transition probability matrix (TPM) Q, as defined in Section 2.1. Each walker performs exactly t steps before yielding a sample. The number of walkers tends to infinity.*

Before we talk about the convergence properties of this Markov chain, we need some definitions.

DEFINITION 12. *The χ^2-contrast $\chi^2(\alpha;\beta)$ of α with respect to β is defined by*

$$\chi^2(\alpha;\beta) = \sum_i \frac{(\alpha(i)-\beta(i))^2}{\beta(i)}.$$

LEMMA 13. *Let $\pi \in \Delta^{n-1}$ a distribution satisfying $\min_i \pi(i) \geq \frac{c}{n}$ for constant $c \leq 1$, and let $u \in \Delta^{n-1}$ denote the uniform distribution. Then, $\chi^2(u;\pi) \leq \left(\frac{1-c}{c}\right)$.*

LEMMA 14. *Let π^t denote the distribution of the walkers after t steps. Its χ^2-divergence with respect to the PageRank vector, π, is*

$$\chi^2(\pi^t;\pi) \leq \left(\frac{1-p_T}{p_T}\right)(1-p_T)^t.$$

PROCESS 15 (TRUNCATED GEOMETRIC). *Independent walkers start on nodes selected uniformly at random and perform random walks on the* original *graph. This means that there is no teleportation and the walk is described by the TPM P as defined in Section 2.1. Each walker performs a random number of steps before yielding a sample. Specifically, the number of steps follows a geometric distribution with parameter p_T. Any walkers still active after t steps are stopped and their positions are acquired as samples. This means that the number of steps is given by the minimum of t and a geometric random variable with parameter p_T. The number of walkers tends to infinity.*

LEMMA 16. *The samples acquired from Process 11 and Process 15 follow the exact same distribution.*

LEMMA 17 (MIXING LOSS). *Let $\pi^t \in \Delta^{n-1}$ denote the distribution of the samples acquired through Process 15. The mass it captures (Definition 2) is lower-bounded as follows.*

$$\mu_k(\pi^t) \geq \mu_k(\pi) - \sqrt{\frac{(1-p_T)^{t+1}}{p_T}}$$

LEMMA 18 (SAMPLING LOSS). *Let $\hat{\pi}_N$ be the estimator of Definition 5 using N samples from the* FROGWILD! *system. This is essentially, Process 15 with the added complication of* random synchronization *as explained in Section 2.2. Also, let π^t denote the sample distribution after t steps, as defined in Lemma 16. The mass captured by this process is lower bounded as follows, with probability at least $1-\delta$.*

$$\mu_k(\hat{\pi}_N) \geq \mu_k(\pi^t) - \sqrt{\frac{k}{\delta}\left[\frac{1}{N} + (1-p_s^2)p_\cap(t)\right]},$$

Combing the results of Lemma 17 and Lemma 18, we establish the main result, Theorem 1.

B.2 Proof of Theorem 2

PROOF. Let $u \in \Delta^{n-1}$ denote the uniform distribution over $[n]$, i.e. $u_i = 1/n$. The two walks start from the same initial uniform distribution, u, and independently follow the same law, Q. Hence, at time t they have the same marginal distribution, $p^t = Q^t u$. From the definition of the augmented transition probability matrix, Q, in Definition 1, we get that

$$\pi_i \geq \frac{p_T}{n}, \quad \forall i \in [n].$$

Equivalently, there exists a distribution $q \in \Delta^{n-1}$ such that

$$\pi = p_T u + (1-p_T)q.$$

Now using this, along with the fact that π is the invariant distribution associated with Q (i.e. $\pi = Q^t\pi$ for all $t \geq 0$) we get that for any $t \geq 0$,

$$\begin{aligned}\|\pi\|_\infty &= \|Q^t\pi\|_\infty \\ &= \|Q^t p_T u + Q^t(1-p_T)q\|_\infty \\ &\geq p_T\|Q^t u\|_\infty.\end{aligned}$$

For the last inequality, we used the fact that Q and q contain non-negative entries. Now we have a useful upper bound for the maximal element of the walks' distribution at time t.

$$\|p^t\|_\infty = \|Q^t u\|_\infty \leq \frac{\|\pi\|_\infty}{p_T} \tag{5}$$

Let M_t be the indicator random variable for the event of a meeting at time t.

$$M_t = \mathbb{I}_{\{\text{walkers meet at time } t\}}$$

Then, $\mathbb{P}(M_t = 1) = \sum_{i=1}^n p_i^t p_i^t = \|p^t\|_2^2$. Since p^0 is the uniform distribution, i.e. $p_i^0 = \frac{1}{n}$ for all i, then $\|p^0\|_2^2 = \frac{1}{n}$. We can also bound the l_2 norm of the distribution at other times. First, we upper bound the l_2 norm by the l_∞ norm.

$$\|p\|_2^2 = \sum_i p_i^2 \leq \sum_i p_i\|p\|_\infty = \|p\|_\infty$$

Here we used the fact that $p_i \geq 0$ and $\sum p_i = 1$.

Now, combining the above results, we get

$$\begin{aligned}p_\cap(t) &= \mathbb{P}\left(\sum_{\tau=0}^t M_\tau \geq 1\right) \leq \mathbb{E}\left[\sum_{\tau=0}^t M_\tau\right] = \sum_{\tau=0}^t \mathbb{E}[M_\tau] \\ &= \sum_{\tau=0}^t \mathbb{P}(M_\tau = 1) = \sum_{\tau=0}^t \|p^\tau\|_2^2 \leq \sum_{\tau=0}^t \|p^\tau\|_\infty \\ &\leq \frac{1}{n} + \frac{t\|\pi\|_\infty}{p_T}.\end{aligned}$$

For the last inequality, we used (5) for $t \geq 1$ and $\|p^0\|_2^2 = 1/n$. This proves the theorem statement. □