FAST-PPR: Scaling Personalized PageRank Estimation for Large Graphs

Peter Lofgren
Department of Computer Science
Stanford University
plofgren@cs.stanford.edu

Siddhartha Banerjee
Department of MS&E
Stanford University
sidb@stanford.edu

Ashish Goel
Department of MS&E
Stanford University
ashishg@stanford.edu

C. Seshadhri
Sandia National Labs
Livermore, CA
scomand@sandia.gov

ABSTRACT

We propose a new algorithm, FAST-PPR, for the Significant-PageRank problem: given input nodes s, t in a directed graph and threshold δ, decide if the Personalized PageRank from s to t is at least δ. Existing algorithms for this problem have a running-time of $\Omega(1/\delta)$; this makes them unsuitable for use in large social-networks for applications requiring values of $\delta = O(1/n)$. FAST-PPR is based on a bi-directional search and requires no preprocessing of the graph. It has a provable average running-time guarantee of $\tilde{O}(\sqrt{d/\delta})$ (where d is the average in-degree of the graph). We complement this result with an $\Omega(1/\sqrt{\delta})$ lower bound for Significant-PageRank, showing that the dependence on δ cannot be improved.

We perform a detailed empirical study on numerous massive graphs showing that FAST-PPR dramatically outperforms existing algorithms. For example, with target nodes sampled according to popularity, on the 2010 Twitter graph with 1.5 billion edges, FAST-PPR has a 20 factor speedup over the state of the art. Furthermore, an enhanced version of FAST-PPR has a 160 factor speedup on the Twitter graph, and is at least 20 times faster on all our candidate graphs.

Keywords

Personalized PageRank, Bi-directional Graph Search, Social Search

1. INTRODUCTION

The success of modern networks is largely due to the ability to search effectively on them. A key primitive behind this is PageRank [1], which is widely used as a measure of network centrality/importance. The popularity of PageRank is in large part due to its fast computation in large networks. As modern social network applications shift towards being more customized to individuals, there is a need for ego-centric measures of network structure.

Personalized PageRank (PPR) [1] has long been viewed as the appropriate ego-centric equivalent of PageRank. For a node u, the personalized PageRank vector π_u measures the frequency of visiting other nodes via short random-walks from u. This makes it an ideal metric for *social search*, giving higher weight to content generated by nearby users in the social graph. Social search protocols find widespread use – from personalization of general web searches [1, 2, 3], to more specific applications like collaborative tagging networks [4], ranking name search results on social networks [5], social Q&A sites [6], etc. The critical primitive for PPR-based social search is the *Significant-PPR* query. For threshold $\delta > 0$, we wish to determine:

Given source node s and target node t, is $\pi_s(t) > \delta$?

More generally, we want an estimator $\widehat{\pi}_s(t)$ which, for some tolerance $\lambda > 0$, is within $(1 \pm \lambda)\pi_s(t)$ for all (s,t) pairs with $\pi_s(t) > \delta$. Current techniques used for PPR estimation (see Section 2.1) have $\Omega(1/\delta)$ running-time – this makes them infeasible for large networks when $\delta = O(1/n)$ or $O(\log(n)/n)$.

In addition to social-search, PPR is also used for a variety of other tasks across different domains: friend recommendation on Facebook [7], who to follow on Twitter [8], graph partitioning [9], community detection [10], and other applications [11]. Other measures of personalization, such as personalized SALSA and SimRank [12], can be reduced to PPR. However, in spite of a rich body of existing work [2, 13, 9, 14, 15, 16, 17], estimating PPR is often a bottleneck in large networks.

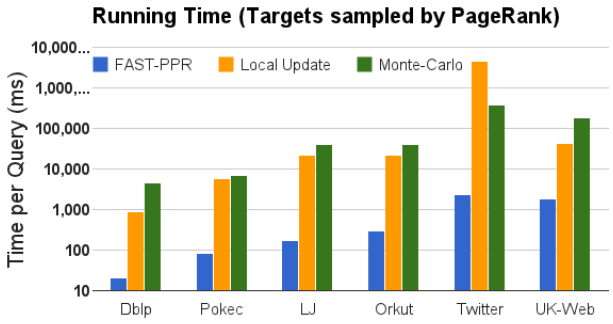
1.1 Our Contributions

We develop a new algorithm, *Frontier-Aided Significance Thresholding for Personalized PageRank* (FAST-PPR), which is based on a new bi-directional search technique for personalized PageRank estimation:

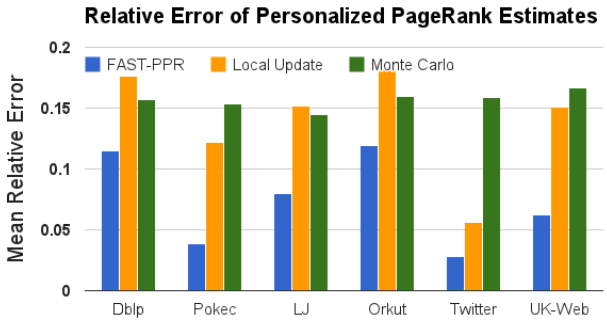
• **Practical Contributions:** We present a simple implementation of FAST-PPR which requires no pre-processing and has an amortized running-time of $\tilde{O}(\sqrt{d/\delta})$[1]. We also propose a simple heuristic, Balanced FAST-PPR, which, by tuning FAST-PPR to balance the forward/reverse work, achieves a significant speedup .

In experiments, FAST-PPR significantly outperforms existing algorithms across a variety of real-life networks. For example, in Figure 1, we compare the running-times and accuracies of FAST-PPR with existing methods. Over a va-

[1]We use $\tilde{O}$-notation for brevity, suppressing poly-log factors, and also dependence on teleport probability α – complete scaling details are provided in our Theorem statements.



(a) Running time (in log scale) of different algorithms



(b) Relative accuracy of different algorithms

Figure 1: Comparison of Balanced FAST-PPR, Monte-Carlo and Local-Update algorithms in different networks – 1000 source-target pairs, threshold $\delta = \frac{4}{n}$, teleport probability $\alpha = 0.2$. Notice that Balanced FAST-PPR is 20 times faster in all graphs, without sacrificing accuracy. For details, see Section 6.

riety of data sets, FAST-PPR is both significantly faster and more accurate than the state of the art.

To give a concrete example: in experiments on the Twitter 2010 graph [18], Balanced FAST-PPR takes less than 3 seconds for answering Significant-PPR queries for random source-target pairs. In contrast, Monte Carlo takes more than 6 minutes and Local Update takes more than an hour. More generally in all graphs, FAST-PPR is at least 15 times faster than the state-of-the-art, without sacrificing accuracy.

• **Theoretical Novelty:** FAST-PPR is the first algorithm for Significant-PPR queries with $\tilde{O}(\sqrt{d/\delta})$ amortized running-time, where $d = m/n$ is the average in-degree[2]. Further, we modify FAST-PPR to get $\tilde{O}(1/\sqrt{\delta})$ worst-case running-time, by pre-computing and storing some additional information, with a required storage of $\tilde{O}(m/\sqrt{\delta})$.

We also give a new running-time *lower bound* of $\Omega(1/\sqrt{\delta})$ for Significant-PPR, which essentially shows that the dependence of FAST-PPR running-time on δ cannot be improved.

Finally, we note that FAST-PPR has the same performance gains for computing PageRank with arbitrary *preference vectors* [2]), wherein the source is picked from a distribution over nodes. Different preference vectors are used for various applications [2, 1]. However, for simplicity, we focus only on personalized PageRank in this work.

2. PRELIMINARIES

We are given underlying directed graph $G(V, E)$, with $|V| = n, |E| = m$ and adjacency matrix A. For any node $u \in V$, we denote $\mathcal{N}^{in}(u), d^{in}(u)$ as the out-neighborhood and out-degree respectively; similarly $\mathcal{N}^{in}(u), d^{in}(u)$ are the in-neighborhood/in-degree. We define $d = \frac{m}{n}$ to be the average in-degree (equivalently, average out-degree).

The personalized PageRank vector π_u for a node $u \in V$ is the stationary distribution of a random walk starting from u, which at each step, with probability α returns to u, and otherwise, moves to a random out-neighbor of the current node. Defining $D = \text{diag}(d^{out}(u))$, and $W = D^{-1}A$, the *personalized PageRank* (PPR) vector of u is given by:

$$\pi_u^T = \alpha \mathbf{e}_u^T + (1-\alpha)\pi_u^T.W, \tag{1}$$

where $\mathbf{e}_u$ is the identity vector of u. Also, for a target node t, we define the *inverse-PPR* of a node w with respect to t as $\pi_t^{-1}(w) = \pi_w(t)$. The inverse-PPR vector $\{\pi_t^{-1}(w)\}_{w \in V}$ of t sums to $n\pi(t)$, where $\pi(t)$ is the global PageRank of t.

Note that the PPR for a uniform random pair of nodes is $1/n$ – thus to identify a pair (s, t) with significant PPR, we need to consider δ of the form $O(1/n)$ or $O(\log n/n)$. In order to process real-time Significant-PPR queries, one option would be precompute all answers, and store a list of all significant targets at each node u. This results in $O(1)$ running-time at query, but requires $\Omega(1/\delta)$ storage per node, which is not feasible in large networks. The other extreme is to eschew storage and compute the PPR at query time – however, as we discuss below, existing algorithms for computing PPR have a worst-case running-time of $\Omega(1/\delta)$. For reasonable values of δ, this is again infeasible.

2.1 Existing Approaches for PPR Estimation

There are two main techniques used to compute PageRank/PPR vectors. One set of algorithms use the power iteration. Since performing a direct power iteration may be infeasible in large networks, a more common approach is to use local-update versions of the power method – this technique was first proposed by Jeh and Widom [2], and subsequently improved by other researchers [19, 9]. These algorithms are primarily based on the following recurrence relation for π_u:

$$\pi_u^T = \alpha e_u^T + \frac{(1-\alpha)}{|\mathcal{N}^{out}(u)|} \cdot \sum_{v \in \mathcal{N}^{out}(u)} \pi_v^T \tag{2}$$

Another use of such local update algorithms is for estimating the inverse-PPR vector for a target node. Local-update algorithms for inverse-PageRank are given in [14] (where inverse-PPR is referred to as the *contribution PageRank vector*), and [20] (where it is called *susceptibility*).

Equation 2 can also be derived from a probabilistic reinterpretation of PPR, which leads to an alternate set of

[2]Formally, for (s, t) with $\pi_s(t) > \delta$, we get an estimate $\widehat{\pi}_s(t)$ with relative error c using $O\left(\frac{1}{c^2}\sqrt{\frac{d}{\delta}}\sqrt{\frac{\log(1/p_{fail})\log(1/\delta)}{\alpha^2 \log(1/(1-\alpha))}}\right)$ runtime; see Corollary 2 in Appendix A for details.

Monte-Carlo algorithms. Given any random variable L taking values in $\mathbb{N}_0$, let $RW(u, L) \triangleq \{u, V_1, V_2, \ldots, V_L\}$ be a random-walk of length $L \sim Geom(\alpha)$[3] starting from u. We have the following alternate characterization of PPR:

$$\pi_u(v) = \mathbb{P}[V_L = v] = \alpha \mathbb{E}\left[\sum_{i=0}^{L} \mathbb{1}_{\{V_i = v\}}\right]. \quad (3)$$

In other words, $\pi_u(v)$ is the probability that v is the last node in $RW(u, L)$, or alternately, proportional to the number of times $RW(u, L)$ visits node v. Both can be used to estimate $\pi_u(\cdot)$ via Monte Carlo algorithms, by generating and storing random walks at each node [13, 15, 16, 17]. Such estimates are easy to update in dynamic settings [15]. However, for significant-PPR queries with threshold δ, these algorithms need $\Omega(1/\delta)$ random-walk samples.

2.2 Intuition for our approach

The problem with the basic Monte Carlo procedure – generating random walks from s and estimating the distribution of terminal nodes – is that to estimate a PPR which is $O(\delta)$, we need $\Omega(1/\delta)$ walks. To circumvent this, we introduce a new *bi-directional estimator* for PPR: given significant-PPR query (s, t, δ), we first work backward from t to find a suitably large set of 'targets', and then do random walks from s to test for hitting this set.

Our algorithm can be best understood through an analogy with the shortest path problem. In the bidirectional shortest path algorithm, to find a path of length l from node s to node t, we find all nodes within distance $\frac{l}{2}$ of t, find all nodes within distance $\frac{l}{2}$ of s, and check if these sets intersect. Similarly, to test if $\pi_s(t) > \delta$, we find all w with $\pi_w(t) > \sqrt{\delta}$ (we call this the *target set*), take $O(1/\sqrt{\delta})$ walks from the start node, and see if these two sets intersect. It turns out that these sets might not intersect even if $\pi_s(t) > \delta$, so we go one step further and consider the *frontier set* – nodes outside the target set which have an edge into the target set. We can prove that if $\pi_s(t) > \delta$ then random walks are likely to hit the frontier set.

Now we describe how our method estimates PPR. Consider a random walk from s to t. At some point, it must enter the frontier. We can decompose the probability of the walk reaching t into the product of two probabilities: the probability that it reaches a node w in the frontier, and the probability that it reaches t starting from w. The two probabilities in this estimate are typically much larger than the overall probability of a random walk reaching t from s, so they can be estimated more efficiently. Fig. 2 illustrates this bi-directional search idea.

2.3 Additional Definitions

To formalize this, we first define a set B to be a blanket set for t with respect to s if all random walks from s to t pass through B. Given blanket set B and a random walk $RW(s, L)$, let H_B be the first node in B hit by the walk (with $H_B = \{\}$ if the walk does not hit B before terminating). From the memoryless property of the geometric random variable, we have:

$$\pi_s(t) = \sum_{w \in B} \mathbb{P}[H_B = w] \cdot \pi_w(t), \quad (4)$$

[3]i.e., $\mathbb{P}[L = i] = \alpha(1-\alpha)^i \, \forall \, i \in \mathbb{N}_0$

i.e., the PPR from s to t is the sum, over all nodes w in the blanket set, of the probability of a random walk hitting the blanket set at node w, times the PPR from w to t.

Recall we define the *inverse-PPR vector* of t as $\pi_t^{-1} = (\pi_w(t))_{w \in V}$. Now we define:

DEFINITION 1 (TARGET SET). *The target set $T_t(\epsilon_r)$ for a target node t is given by:*

$$T_t(\epsilon_r) := \{w \in V : \pi_t^{-1}(w) > \epsilon_r\}.$$

DEFINITION 2 (FRONTIER SET). *The frontier set $F_t(\epsilon_r)$ for a target node t is defined as:*

$$F_t(\epsilon_r) := \{w \in V \setminus T_t(\epsilon_r) : \exists x \in T_t(\epsilon_r) \text{ s.t. } (w, x) \in E\}.$$

In other words, the target set $T_t(\epsilon_r)$ contains all nodes with inverse-PPR greater than ϵ_r, while the frontier set $F_t(\epsilon_r)$ contains all nodes which are in-neighbors of $T_t(\epsilon_r)$, but not in $T_t(\epsilon_r)$. The threshold ϵ_r is chosen later, but for the moment we assume $\epsilon_r = \sqrt{\delta}$.

2.4 A Warm-up Proposition

The next proposition illustrates the core of our approach:

PROPOSITION 1. *For any pair $(s, t) \in V^2$, and threshold $\epsilon_r < \alpha$:*

1. *The frontier set $F_t(\sqrt{\delta})$ is a blanket set of t with respect to any source node $s \notin T_t$.*
2. *For random-walk $RW(s, L)$ from some $s \notin T_t(\sqrt{\delta})$, and with length $L \sim Geom(\alpha)$, we have:*

$$\mathbb{P}[RW(s, L) \text{ hits } F_v(\sqrt{\delta})] \geq \frac{\pi_s(t)}{\epsilon_r}$$

PROOF. For brevity, we write $T_t = T_t(\sqrt{\delta}), F_t = F_t(\sqrt{\delta})$. For the first part, note that from the definition of PPR, we have $\pi_t(t) \geq \alpha$. Thus, given that $\epsilon_r < \alpha$, we have that $t \in T_t$. Further, the frontier set F_t contains all neighbors of nodes in T_t which are not themselves in T_t. Hence, given any source node $u \notin T_t$, for a random walk from s to reach t, it must first hit a node in F_t.

For the second part, since F_t is a blanket set for $u \notin T_v$ with respect to target t, we have from equation 4 that:

$$\pi_s(t) = \sum_{w \in F_t} \mathbb{P}[H_{F_t} = w] \, \pi_w(t),$$

where H_{F_t} is the first node where the walk hits F_t. Further, if $w \in F_t$, then by definition, $w \notin T_t$ and hence $\pi_w(t) \leq \epsilon_r \, \forall \, w \in F_t$. Substituting, we get:

$$\pi_s(t) \leq \epsilon_r \sum_{w \in F_t} \mathbb{P}[H_{F_t} = w] = \epsilon_r \mathbb{P}[H_{F_t} \neq \phi].$$

Rearranging, we get the result. □

Using Proposition 1, we can reduce any significant-PPR query (s, t, δ) to estimating the probability of random walks from s hitting $F_t(\sqrt{\delta})$ – since this probability is at least $\sqrt{\delta}$ for pair (s, t) *s.t.* $\pi_s(t) \geq \delta$, we only need $\tilde{O}(1/\sqrt{\delta})$ walks to get an estimator that accepts (s, t) with high probability.

We note that similar bi-directional techniques have been used for estimating the mixing time of *fixed length* random walks in *regular undirected graphs* [21, 22]. These results however do not extend to estimating PPR – in particular,

Algorithm 1 FAST-PPR(s, t, δ)

Inputs: start node s, target node t, threshold δ, reverse threshold ϵ_r (default value $\sqrt{\delta}$), failure probability p_{fail} (default value 10^{-6})

1: Define $\epsilon_f = \delta/\epsilon_r$
2: Call FRONTIER(t, ϵ_r) to obtain sets $T_t(\epsilon_r), F_t(\epsilon_r)$, inverse-PPR values $(\pi_t^{-1}(w))_{w \in F_t(\epsilon_r)}$.
3: **if** $s \in T_t(\epsilon_r)$ **then**
4: Accept (s, t) as significant
5: **else**
6: Set $n_f = \frac{24 \log(1/p_f)}{\epsilon_f}$ (See Theorem 2 for details)
7: **for** index $i \in [n_f]$ **do**
8: Generate $L_i \sim Geom(\alpha)$
9: Generate random-walk $RW_i(s, L_i)$
10: Determine H_i, the first node in $F_t(\epsilon_r)$ hit by RW_i; if RW_i never hits $F_t(\epsilon_r)$, set $H_i = \{\}$
11: **end for**
12: Compute $\widehat{\pi}_s(t) = \frac{1}{n_f} \sum_{i \in [n_f]} \pi_t^{-1}(H_i)$
13: Accept (s, t) if $\widehat{\pi}_s(t) > \delta$, else reject
14: **return** $\widehat{\pi}_s(t)$
15: **end if**

Algorithm 2 FRONTIER(t, ϵ_r)

Inputs: target node t, reverse threshold ϵ_r, multiplicative factor β (default= 1/6)

1: Define additive error $\epsilon_{inv} = \beta \epsilon_r$
2: Initialize (sparse) estimate-vector $\widehat{\pi}_t^{-1}$ and (sparse) residual-vector r_t as: $\begin{cases} \widehat{\pi}_t^{-1}(u) = r_t(u) = 0 \text{ if } u \neq t \\ \widehat{\pi}_t^{-1}(t) = r_t(t) = \alpha \end{cases}$
3: Initialize target-set $\widehat{T}_t = \{t\}$, frontier-set $\widehat{F}_t = \{\}$
4: **while** $\exists w \in V \; s.t. \; r_t(w) > \alpha \epsilon_{inv}$ **do**
5: **for** $u \in \mathcal{N}^{in}(w)$ **do**
6: $\Delta = (1 - \alpha) . \frac{r_t(w)}{d^{out}(u)}$
7: $\widehat{\pi}_t^{-1}(u) = \widehat{\pi}_t^{-1}(u) + \Delta, r_t(u) = r_t(u) + \Delta$
8: **if** $\widehat{\pi}_t^{-1}(u) > \epsilon_r$ **then**
9: $\widehat{T}_t = \widehat{T}_t \cup \{u\}$, $\widehat{F}_t = \widehat{F}_t \cup \mathcal{N}^{in}(u)$
10: **end if**
11: **end for**
12: $r_t(w) = 0$
13: **end while**
14: $\widehat{F}_t = \widehat{F}_t \setminus \widehat{T}_t$
15: Run DE-BIAS$(\widehat{\pi}_t^{-1}, \widehat{T}_t, \epsilon_{inv})$
16: **return** $\widehat{T}_t, \widehat{F}_t, (\widehat{\pi}_t^{-1}(w))_{w \in \widehat{F}_t \cup \widehat{T}_t}$

we need to consider *directed graphs, arbitrary node degrees and walks of random length.* Also, a bi-directional scheme was proposed by Jeh and Widom [2], who suggested using a *fixed skeleton* of target nodes for estimating *PageRank*. However there are no provable running-time guarantees for such schemes; also, the hubs skeleton and partial vectors need to be pre-computed and stored. We note that our scheme uses *separate target sets for each target node.*

3. THE FAST-PPR ALGORITHM

Building on the intuition presented above, we now develop the *Frontier-Aided Significance Thresholding* algorithm, or FAST-PPR, which is specified in Algorithm 1. Instead of estimating the probability of hitting the frontier set, Algorithm 1 uses a sharper estimate based on the inverse-PPR of nodes in the frontier set. This lets us control false positives.

For our bi-directional search, we need sets $T_t(\epsilon_r), F_t(\epsilon_r)$ and inverse-PPR values $(\pi_t^{-1}(w))_{w \in F_t(\epsilon_r)}$. One way to get this is by storing the information as oracles at each node – we discuss this implementation in Section 4.1.

For Algorithm 1 however, we use a local-update procedure FRONTIER that generates an *approximate* inverse-PPR estimate. FRONTIER, which is specified in Algorithm 2, is based on existing algorithms for inverse-PPR [14, 20]. It works by distributing some weight among nodes, while maintaining an invariant based on Equation 2.

The local-update algorithms of [14, 20] result in an inverse-PPR estimate which has a bounded negative bias – $\forall w$, they guarantee $\pi_t^{-1}(w) - \widehat{\pi}_t^{-1}(w) \in [0, \epsilon_{inv}]$. To remove this, we use a procedure DE-BIAS (Step 15 of Algorithm 2, which we describe in more detail in Section 4.2). We note however that this does not improve the empirical accuracy significantly, and so can be omitted in practice – see Section 6.3 for details regarding this.

Characterizing the accuracy of the FAST-PPR algorithm is somewhat involved, and we defer the formal analysis to Section 4. However, it is easy to characterize the running-time of FAST-PPR with FRONTIER. In particular, this can be decomposed into two parts: the *reverse time* for estimating the inverse-PPR via the FRONTIER algorithm, and the *forward time* spent executing random walks from the start node. Algorithm 1 has two parameters, ϵ_f and ϵ_r, which can be used to control these running-times, as follows:

THEOREM 1. *Suppose we are given threshold δ and desired failure probability p_f. Then for each Significant-PPR query, the FAST-PPR algorithm with parameters ϵ_r, ϵ_f (s.t. $\epsilon_r \epsilon_f \leq \delta$), and n_f chosen as in Algorithm 2, requires:*

- *(Average) Reverse time* $= O\left(\frac{d}{\alpha \epsilon_r}\right)$
- *Forward time* $= O\left(\frac{\log(1/p_f)}{\alpha \epsilon_f}\right)$,

where $d = \frac{m}{n}$ is the average in-degree of the network. Furthermore, given pre-computation time and additional storage of $O(\frac{d}{\alpha \epsilon_r})$ per node, each query requires $O\left(\frac{\log(1/p_f)}{\alpha \epsilon_f}\right)$ running time in the worst-case.

PROOF OF THEOREM 1. The guarantee for the forward time follows from our choice of n_f in Theorem 2, and the ob-

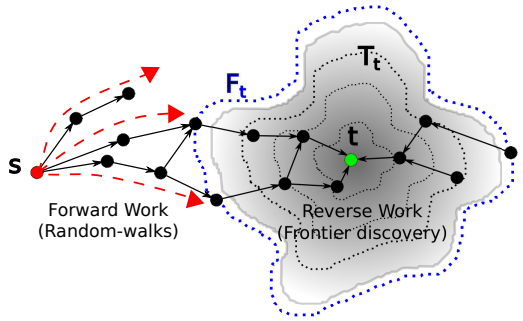


Figure 2: The FAST-PPR Algorithm: We first work backward from the target t to find the frontier F_t (with inverse-PPR estimates). Next we work forward from the source s, generating random-walks and testing for hitting the frontier.

servation that each walk $RW(u, Geom(\alpha))$ on average takes $1/\alpha$ steps. The reverse time guarantee follows from existing running-time bounds for local-update algorithms for inverse-PPR – for example, see Theorem 2 in [20]. For worst-case runtime, we can precompute and store the frontier for all nodes, and only perform random walks at runtime. The required pre-computation time is the same as above; for the storage requirement, observe that for any node $w \in V$, it can belong to the target set of at most $\frac{1}{\epsilon_r}$ nodes, as $\sum_{t \in V} \pi_t^{-1}(w) = 1$. Summing over all nodes, we have:

$$\begin{aligned}\text{Total Storage} &\leq \sum_{t \in V} \sum_{w \in T_t} \sum_{u \in \mathcal{N}^{in}(w)} \mathbb{1}_{\{u \in F_t\}} \\ &\leq \sum_{w \in V} \sum_{t \in V : w \in T_t} d^{in}(w) \leq \frac{m}{\epsilon_r}\end{aligned}$$

□

For the worst-case runtime in the above result, we precompute and store the frontier, and generate random-walks at runtime. In addition, we can also pre-compute and store the random-walks from all nodes, and perform appropriate joins at runtime to get the FAST-PPR estimate. This allows us to implement FAST-PPR on any distributed system that can do fast intersections/joins. The above theorem also suggests how we can balance the forward and reverse times – in particular, we have the following corollary:

COROLLARY 1. *Let* $\epsilon_f = \sqrt{\frac{\delta \log(1/p_f)}{d}}, \epsilon_r = \sqrt{\frac{d\delta}{\log(1/p_f)}}$, *where* $d = m/n$. *Then FAST-PPR with FRONTIER has an average per-query running-time of* $O\left(\frac{1}{\alpha}\sqrt{\frac{d \log(1/p_{fail})}{\delta}}\right)$, *or alternately, worst-case running time and per-node storage of* $O\left(\frac{1}{\alpha}\sqrt{\frac{d \log(1/p_{fail})}{\delta}}\right)$.

Note that the reverse time bound in Theorem 1 is amortized across all choices of target nodes; for some target nodes (those with high global PageRank) the reverse time will much larger than average, and for others it will be much lower than average. On the other hand, the forward time is the same for all source nodes and is very predictable.

3.1 Balanced FAST-PPR

To improve performance in practice, we can optimize the choice of ϵ_r and ϵ_f based on the target t to balance the reverse and forward time. Notice for each value of ϵ_r, we can compute:

$$n_f(\epsilon_r) = \frac{24(\log(1/p_f))\epsilon_r}{\delta},$$

and accurately estimate the forward time based on $n_f(\epsilon_r)$. Thus instead of committing to a single value of ϵ_r, in the balanced version of FAST-PPR we run the while loop in FRONTIER a varying number of iterations until the estimated remaining forward time equals the reverse time already spent. Since we do not have a fixed additive error target ϵ_{inv}, we must use a version of FRONTIER using a priority queue on residual values. In the while loop, instead of choosing any w such that $r_t(w) > \alpha\epsilon_{inv}$, we choose the node w with the greatest value of $r_t(w)$. We can also compute the current additive error guarantee ϵ_{inv} to be $\frac{1}{\alpha}$ times the current largest residual value. This version of the algorithm is described in more detail in [20].

In Balanced FAST-PPR, we expand the frontier until the reverse-time spent exceeds our estimate for the required forward-time, $c_{\text{walk}} \cdot n_f(\epsilon_r)$, where $\epsilon_r = \frac{\epsilon_{inv}}{\beta}$, ϵ_{inv} is the current additive error guarantee, and c_{walk} is the empirical time it takes to generate a walk. In Section 6.4, we show how this change balances forward and reverse work and significantly reduces the average running-time.

4. THEORETICAL BASIS OF FAST-PPR

We now characterize the accuracy of the FAST-PPR algorithm. We first consider an idealized version of the FAST-PPR algorithm, where the frontier for each node is *precomputed and stored as an oracle*. Subsequently, we consider the case of imperfect oracles, where we discuss how the inverse-PPR estimates of FRONTIER have a persistent negative bias, which may cause the FAST-PPR estimates to be inaccurate. We show how this bias can be formally removed to get a theoretical guarantee, and also propose a heuristic procedure, which is simpler and works well in practice.

4.1 FAST-PPR with Perfect Frontier-Estimates

In this section, we assume that for any target node t, the target and frontier sets (T_t, F_t) and also the corresponding inverse-PPR values $(\pi_v^{-1}(w))_{w \in F_t \cup T_t}$ are stored as oracles. We analyze this Oracle-FAST-PPR algorithm primarily for ease of presentation – however, storing such oracles may prove useful in some settings. In particular, storage allows us to get *worst-case* theoretical bounds on the running-time:

THEOREM 2 (CORRECTNESS OF ORACLE-FAST-PPR). *We are given Significant-PPR query* (s, t, δ) *and desired failure probability* p_{fail}. *For any* $\lambda \in [0, 1]$, *suppose we choose* ϵ_r, ϵ_f *s.t.* $\epsilon_r, \epsilon_f \in (0, 1), \epsilon_r\epsilon_f = \delta$, *and* $n_f = \frac{3\ln(2/p_{fail})}{\epsilon_f \lambda^2}$. *Then for pair* (s, t) *such that* $\pi_s(t) > \delta$, *with probability at least* $1 - p_{fail}$, *the Oracle-FAST-PPR estimate* $\widehat{\pi}_s(t)$ *obeys:*

$$|\pi_s(t) - \widehat{\pi}_s(t)| \leq \lambda \pi_s(t).$$

Furthermore, given a desired tolerance $c > 1$, *if we choose:*

$$n_f = \left(\frac{3c(c+1)}{(c-1)^2}\right)\frac{\ln(1/p_{fail})}{\epsilon_f}.$$

Then with probability at least $1 - p_{fail}$, *Oracle-FAST-PPR will:*

- *Accept pair* (s, t) *if* $\pi_s(t) > c\delta$
- *Reject pair* (s, t) *if* $\pi_s(t) < \delta/c$

PROOF OF THEOREM 2. Recall we are given the tolerance parameter $c > 1$ and failure probability p_{fail}. We now have two mutually exclusive two cases:

Controlling False Negatives: Suppose $\pi_s(t) \geq c\delta$ – we want to show that with probability at least $1 - p_{fail}$, the estimate $\widehat{\pi}_s(t)$ is greater than δ. From equation 4, we have:

$$\pi_s(t) = \sum_{w \in \widehat{F}_t} \mathbb{P}\left[H_{\widehat{F}_t}(RW(s, L)) = w\right] \pi_t^{-1}(w)$$

On the other hand, our PPR estimate obeys:

$$\widehat{\pi}_s(t) = \frac{1}{n_f} \sum_{i \in [n_f]} \sum_{w \in F_t} \mathbb{1}_{\{H_i = w\}} \pi_t^{-1}(w)$$

Suppose we define $X_i = \sum_{w \in F_t} \mathbb{1}_{\{H_i = w\}} \pi_t^{-1}(w)$ – in other words, for random-walk RW_i, we set X_i to be the inverse-PPR of the first node in F_t it hits, and set $X_i = 0$ if the

walk terminates before hitting F_t . Then clearly $\{X_i\}_{i\in[n_f]}$ are i.i.d random variables, with mean $\mathbb{E}[X_i] = \pi_s(t) > \delta$. Further, since for any node $w \in F_t$, we have $\pi_t^{-1}(w) \leq \epsilon_r$, this implies that $X_i \in [0, \epsilon_r]$. Now if $Y_i = X_i/\epsilon_r$, then clearly $Y_i \in [0, 1]$, and also $\mathbb{E}[Y_i] \geq \delta/\epsilon_r = \epsilon_f$. Let $Y = \sum_{i\in[n_f]} Y_i$; then $\mathbb{E}[Y] \geq n_f\epsilon_f$.

Now, using a standard Chernoff Bound (for example, see Chapter 4 of [23]), we have for any $\lambda \in [0, 1]$:

$$\begin{aligned}\mathbb{P}\left[|\widehat{\pi}_s(t) - \pi_s(t)| > \lambda\pi_s(t)\right] &\leq \mathbb{P}\left[|Y - \mathbb{E}[Y]| > c\mathbb{E}[Y]\right] \\ &\leq 2\exp\left(-\frac{\lambda^2\mathbb{E}[Y]}{3}\right) \\ &\leq 2\exp\left(-\frac{\lambda^2 n_f\epsilon_f}{3}\right).\end{aligned}$$

Now, if $n_f \geq \frac{3\ln(2/p_{fail})}{\lambda^2\epsilon_f}$, we have $\mathbb{P}\left[\widehat{\pi}_s(t) < \delta\right] \leq p_{fail}$. Finally, if $\pi_s(t) > c\delta$, we can choose $\lambda = 1 - 1/c$ to get the desired control for false negatives.

Controlling False Positives: Suppose now $\pi_s(t) < \frac{\delta}{c}$. In this case, we want to show that with probability at least $1 - p_{fail}$, the estimate $\widehat{\pi}_s(t)$ is less than δ.

We define X_i, Y_i and Y as above – note we again have $Y_i \in [0, 1]\,\forall\, i$, and also $\mathbb{E}[Y] = n_f\pi_s(t)/\epsilon_r \leq n_f\epsilon_f/c$. Then:

$$\mathbb{P}\left[\widehat{\pi}_s(t) > \delta\right] = \mathbb{P}\left[Y > n_f\epsilon_f\right] = \mathbb{P}\left[Y > (1+\lambda)\mathbb{E}[Y]\right],$$

where $\lambda = \frac{n_f\epsilon_f}{\mathbb{E}[Y]} - 1 > c - 1$. Now, using the Chernoff bound for the upper tail, we have:

$$\mathbb{P}\left[\widehat{\pi}_s(t) > \delta\right] \leq \exp\left(-\frac{\lambda^2\mathbb{E}[Y]}{2+\lambda}\right) = \exp\left(-\frac{(n_f\epsilon_f - \mathbb{E}[Y])^2}{\mathbb{E}[Y] + n_f\epsilon_f}\right)$$

Since decay rate of the exponent is decreasing in the range $\mathbb{E}[Y] < n_f\epsilon_f$, using $\mathbb{E}[Y] \leq n_f\epsilon_f/c$, we have:

$$\mathbb{P}\left[\widehat{\pi}_s(t) > \delta\right] \leq \exp\left(-n_f\epsilon_f\frac{(c-1)^2}{c(c+1)}\right)$$

Substituting $n_f = \frac{3c(c+1)\ln(1/p_{fail})}{(c-1)^2\epsilon_f}$ gives $\mathbb{P}\left[\widehat{\pi}_s(t) < \delta\right] \leq p_{fail}$. This completes the proof. □

4.2 Using Approximate Frontier-Estimates

To see how approximations due to FRONTIER affect FAST-PPR, note that if $\epsilon_{inv} = \beta\epsilon_r$, then the additive error guarantee is not useful when applied to nodes in the frontier. For example, we could have $\pi_t^{-1}(w) < \epsilon_{inv}$ for all w in the frontier $\widehat{F}_t(\epsilon_r)$. We give a brief sketch of how we can modify FAST-PPR to account for this – the full algorithm, along with the proof of correctness, is deferred to Appendix A.

First, note that for $w \in \widehat{T}_t(\epsilon_r)$, we have a multiplicative bound (see Lemma 1 in Appendix A): $\widehat{\pi}_t^{-1}(w) \in (1 \pm c_1)\pi_t^{-1}(w)$ for some $c_1 > 0$. Ideally, we want to bootstrap these 'good' estimates in the target-set. However, if we allow walks to reach the target set, then the variance of our estimate is too high (see fig. 5(c) for empirical evidence of this observation). To circumvent this, we re-weight the walks so that they do not enter the target set. Note that for random walk RW_i, whenever we reach a node w in the frontier $\widehat{F}_v(\epsilon_r)$, we can decompose its PPR estimate $\pi_w(t)$ as a sum of terms from two mutually-exclusive events: in the next step, the walk enters the target set, or does not. For the former event, we have a good estimate, with a multiplicative guarantee. To improve our estimate of the latter, we continue the walk, but bias it to only travel to nodes outside the target-set. By adding these de-biasing terms, we get a multiplicative guarantee for any (s, t) such that $\pi_s(t) > \delta$. Refer Appendix A for details.

The additional complexity of the estimator described above is mainly for theoretical purposes. In practice, the basic FAST-PPR algorithm with approximate frontier estimates (Algorithm 1) performs well. Further, its performance can be improved by the following heuristic DE-BIAS procedure:

Algorithm 3 DE-BIAS($\widehat{\pi}_t^{-1}, \widehat{T}_t, \epsilon_{inv}$)

```
for w ∈ T̂_t do
    for u ∈ N^in(w) do
        π̂_t^{-1}(u) = π̂_t^{-1}(u) + ((1−α)/d^out(u)) · (ε_inv/2)
    end for
end for
```

Note that we add a correction-factor of $\frac{\epsilon_{inv}}{2}$ only for nodes in the target set – it is easy to check that the resulting estimates satisfy: $|\pi_t^{-1}(w) - \widehat{\pi}_t^{-1}(w)| < \frac{\epsilon_{inv}}{2}$. Also, DE-BIAS does not change the running-time of FAST-PPR – however, in fig. 5, we show empirically that using the DE-BIAS procedure improves the accuracy of FAST-PPR.

5. LOWER BOUND FOR SIGNIFICANT-PPR

In this section, we prove that any algorithm that accurately answers *Significant-PPR* queries must look at $\Omega(1/\sqrt{\delta})$ edges of the graph. We note that the numerical constants are chosen for easier calculations, and are not optimized.

We assume $\alpha = 1/100\log(1/\delta)$, and consider randomized algorithms for the following variant of Significant-PPR, which we denote as *Detect-High*(δ) – for all pairs (s, t):

- If $\pi_s(t) > \delta$, output ACCEPT with probability $> 9/10$.
- If $\pi_s(t) < \frac{\delta}{2}$, output REJECT with probability $> 9/10$.

We stress that the probability is over the random choices of the algorithm, *not* over s, t. We now have the following lower bound:

THEOREM 3. *Any algorithm for* Detect-High(δ) *must access* $\Omega(1/\sqrt{\delta})$ *edges of the graph.*

PROOF OUTLINE. The proof uses a lower bound of Goldreich and Ron for *expansion testing* [24]. The technical content of this result is the following – consider two distributions $\mathcal{G}_1$ and $\mathcal{G}_2$ of undirected 3-regular graphs on N nodes. A graph in $\mathcal{G}_1$ is generated by choosing three uniform-random perfect matchings of the nodes. A graph in $\mathcal{G}_2$ is generated by randomly partitioning the nodes into 4 equal sets, $V_i, i \in \{1, 2, 3, 4\}$, and then, within each V_i, choosing three uniform-random matchings.

Consider the problem of distinguishing $\mathcal{G}_1$ from $\mathcal{G}_2$. An adversary arbitrarily picks one of these distributions, and generates a graph G from it. A *distinguisher* must report whether G came from $\mathcal{G}_1$ or $\mathcal{G}_2$, and it must be correct with probability $> 2/3$ regardless of the distribution chosen. Theorem 7.5 of Goldreich and Ron [24] asserts the following:

THEOREM 4 (THEOREM 7.5 OF [24]). *Any distinguisher must look at* $\sqrt{N}/5$ *edges of the graph.*

We perform a direct reduction, relating the *Detect-High*(δ) problem to the Goldreich and Ron setting – in particular, we

Table 1: Datasets used in experiments

Dataset	Type	# Nodes	# Edges
DBLP-2011	undirected	986K	6.7M
Pokec	directed	1.6M	30.6M
LiveJournal	undirected	4.8M	69M
Orkut	undirected	3.1M	117M
Twitter-2010	directed	42M	1.5B
UK-2007-05	directed	106M	3.7B

show that an algorithm for *Detect-High*(δ) which requires less than $1/\sqrt{\delta}$ queries can be used to construct a distinguisher which violates Theorem 4. The complete proof is provided in Appendix B. □

6. EXPERIMENTS

We conduct experiments to explore three main questions:

1. How fast is FAST-PPR relative to previous algorithms?
2. How accurate are FAST-PPR's estimates?
3. How is FAST-PPR's performance affected by our design choices: de-biasing, use of frontier and balancing forward/reverse running-time?

6.1 Experimental Setup

• **Data-Sets:** To measure the robustness of FAST-PPR, we run our experiments on several types and sizes of graph, as described in Table 1. Pokec and Twitter are both social networks in which edges are directed. The LiveJournal, Orkut (social networks) and DBLP (collaborations on papers) networks are all undirected – for each, we have the largest connected component of the overall graph. Finally, our largest dataset with 3.7 billion edges is from a 2007 crawl of the UK domain [25, 26]. Each vertex is a web page and each edge is a hyperlink between pages.

For detailed studies of FAST-PPR, we use the Twitter-2010 graph, with 41 million users and 1.5 billion edges. This presents a further algorithmic challenge because of the skew of its degree distribution: the average degree is 35, but one node has more than 700,000 in-neighbors.

The Pokec [27], Live Journal [28], and Orkut [28] datasets were downloaded from the Stanford SNAP project [29]. The DBLP-2011 [25], Twitter-2010 [25] and UK 2007-05 Web Graph [25, 26] were downloaded from the Laboratory for Web Algorithmics [18].

• **Implementation Details:** We ran our experiments on a machine with a 3.33 GHz 12-core Intel Xeon X5680 processor, 12MB cache, and 192 GB of 1066 MHz Registered ECC DDR3 RAM. Each experiment ran on a single core and loaded the graph used into memory before beginning any timings. The RAM used by the experiments was dominated by the RAM needed to store the largest graph using the SNAP library format [29], which was about 21GB.

For reproducibility, our C++ source code is available at: http://www.stanford.edu/~plofgren/fast_ppr/.

• **Benchmarks:** We compare FAST-PPR to two benchmark algorithms: Monte-Carlo and Local-Update.

Monte-Carlo denotes the standard random-walk algorithm [13, 15, 16, 17] – we perform $\frac{c_{MC}}{\delta}$ walks and estimate $\pi_u(v)$ by the fraction of walks terminating at v. For our experiments, we choose $c_{MC} = 35$, to ensure that the relative errors for Monte-Carlo are the same as the relative error bounds chosen for Local-Update and FAST-PPR (see below). However, even in experiments with $c_{MC} = 1$, we find that FAST-PPR is still 3 times faster on all graphs and 25 times faster on the largest two graphs (see Appendix C).

Our other benchmark, Local-Update, is the state-of-the-art local power iteration algorithm [14, 20]. It follows the same procedure as the FRONTIER algorithm (Algorithm 2), but with the additive accuracy ϵ_{inv} set to $\delta/2$. We note here that a backward local-update is more suited to Significant-PPR than a forward scheme [9, 2] as the latter do not have natural performance guarantees on directed graphs.

• **Parameters:** For FAST-PPR, we use failure probability $p_{fail} = 10^{-6}$, and $n_f = 24\ln(p_{fail})/\epsilon_f$. For vanilla FAST-PPR, we use $\epsilon_r = \epsilon_f = \sqrt{\delta}$, while for Balanced FAST-PPR, we use $\epsilon_f = \sqrt{\frac{48\delta\log(1/p_{fail})}{m\pi(v)}}, \epsilon_r = \frac{\delta}{\epsilon_f}$.

6.2 Running Time Comparisons

Our experimental procedure is the following – after loading the graph into memory, we sample 1000 source/target pairs (s,t) uniformly at random. For each, we measure the time required for Significant-PPR queries with threshold $\delta = \frac{4}{n}$. Note that the PPR for two random nodes in an n node graph is $\frac{1}{n}$, so we are testing for a value 4 times greater than average. Because of the skew of the PPR distribution, a value of $\frac{4}{n}$ is fairly significant. For example, in the Twitter graph, 97% of node pairs have PPR less than $\frac{1}{n}$, and 99% have PPR less than $\frac{4}{n}$ (see Appendix C for details).

To keep the experiment length less than 24 hours, for the Local-Update algorithm and Monte-Carlo algorithms we only use 20 and 5 pairs respectively.

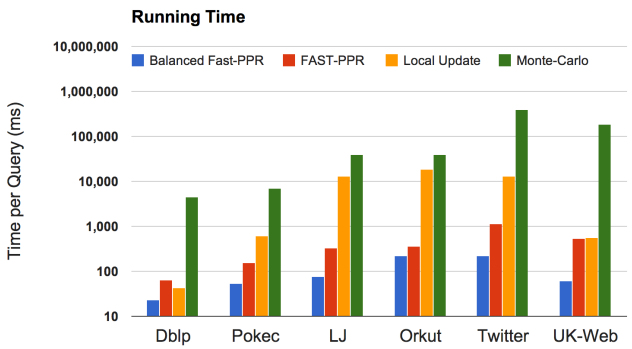
The running-time comparisons are shown in Figure 3 – we compare Monte-Carlo, Local-Update, standard FAST-PPR (Section 3, with $\epsilon_r = \epsilon_f = \sqrt{\delta}$), and also Balanced FAST-PPR, the method we described in Section 3.1.

The speedup is more prominent when target nodes are sampled from the PageRank distribution. This is important, as it is a more realistic model for queries in personalized search applications, with searches biased towards more popular targets. To quantify this effect further, we sort the targets by their global PageRank, choose the first target in each percentile, and measure the running time of the four algorithms, as shown in Figure 4. Note that FAST-PPR is much faster than previous methods for the targets with high PageRank. Note also that large PageRank targets account for most of the average running-time – thus improving performance in these cases causes significant speedups in the average running time.

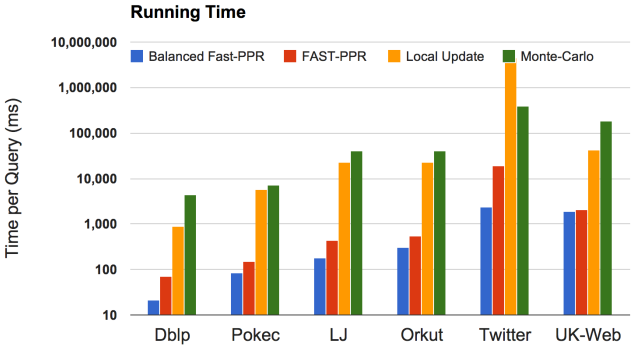
To give a sense of the speedup achieved by FAST-PPR, consider the Twitter-2010 graph. Balanced FAST-PPR takes less than 3 seconds for Significant-PPR queries with targets sampled from global PageRank – in contrast, Monte Carlo takes more than 6 minutes and Local Update takes more than an hour. In the worst-case, Monte Carlo takes 6 minutes and Local Update takes 6 hours, while Balanced FAST-PPR takes 40 seconds. Finally, the estimates from FAST-PPR are twice as accurate as the Local Update, and 6 times more accurate than Monte Carlo.

6.3 Characterizing the Accuracy of FAST-PPR

Next we measure the empirical accuracy of FAST-PPR. For each graph we sample 25 targets uniformly at random, and compute their Inverse-PPR vectors using Local-Update to an additive error of $\frac{\delta}{100}$ (as before, we use $\delta = \frac{4}{n}$). We



(a) Sampling targets uniformly



(b) Sampling targets from PageRank distribution

Figure 3: Average running-time (on log-scale) for different networks. We measure the time required for Significant-PPR queries (s, t, δ) with threshold $\delta = \frac{4}{n}$ for 1000 (s, t) pairs. For each pair, the start node is sampled uniformly, while the target node is sampled uniformly in Figure 3(a), or from the global PageRank distribution in Figure 3(b). In this plot we use teleport probability $\alpha = 0.2$.

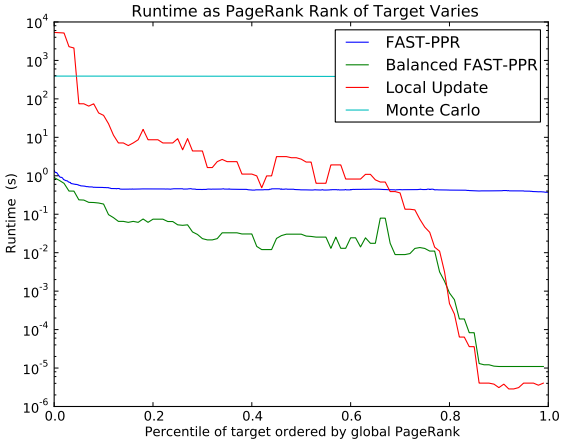


Figure 4: Execution time vs. global PageRank of the target. Target nodes were sorted by global PageRank, then one was chosen from each percentile. We use $\alpha = 0.2$, and 5-point median smoothing.

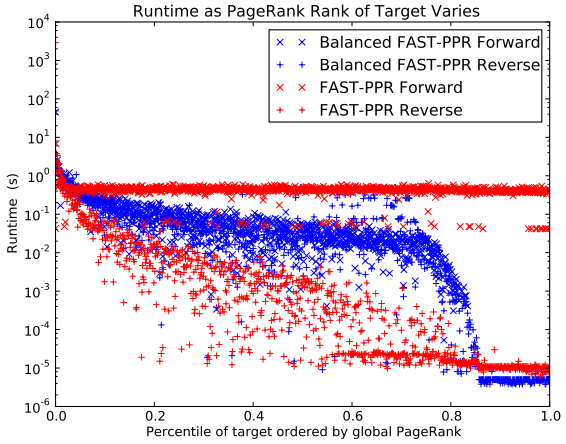


Figure 6: Forward and reverse running-times (on log-scale) for FAST-PPR (in red) and Balanced FAST-PPR (in blue) on the Twitter 2010 graph.

want to measure how well FAST-PPR can distinguish pairs (s, t) such that $\pi_s(t) < \delta$ from pairs with $\pi_s(t) > \delta$. We balance the number of positive and negative examples, and ensure that the source and target pairs have true personalized PageRank within a factor of 4, i.e., for each target t, we sample 50 nodes from the set $\{s : \frac{\delta}{4} \leq \pi_s(t) \leq \delta\}$ and 50 nodes from the set $\{s : \delta \leq \pi_s(t) \leq 4\delta\}$. We execute FAST-PPR for each of the 2500 (s, t) pairs, and measure the empirical error – the results are compiled in Table 2.

To make sure that FAST-PPR is not sacrificing accuracy for improved runtime, we also compare with the relative error of Monte-Carlo and Local-Update, using the same parameters for our running-time experiments. Note it would more than 50 days of computation to run Monte-Carlo for all 2500 pairs. To circumvent this we sample from a Binomial Distribution with $p_s = \pi_s(t)$ for each target t – note that this is statistically equivalent to running Monte-Carlo over all pairs. The relative errors are shown in Figure 5(a).

We also plot the performance of FAST-PPR without the de-biasing step – note that it performs worse than FAST-PPR.

6.4 Some Other Empirical Observations

• **De-Biasing:** Another way to visualize accuracy is by plotting the true PPR vs estimated PPR. In Figure 5(c), we run FAST-PPR without de-biasing – notice that most estimates lie below the line $y = x$. As we discuss in Section 4.2, this is caused by the negative bias in inverse-PPR estimates from FRONTIER. In contrast, note that in Figure 5(b), where we run FAST-PPR with the de-biasing heuristic, the points cluster more tightly around the $y = x$ line.

• **Necessity of the Frontier:** Another question we study is whether we can modify FAST-PPR to compute the target set $T_t(\epsilon_r)$ and then run Monte-Carlo walks until they hit the target set (rather than the frontier set $F_t(\epsilon_r)$). This may appear natural, as we have good control of inverse-PPR values in the target set; further, it would reduce storage requirements for an oracle-based implementation.

Table 2: Accuracy of FAST-PPR (with parameters as specified in Section 6.1)

	Dblp	Pokec	LJ	Orkut	Twitter	UK-Web
Threshold δ	4.06e-06	2.45e-06	8.25e-07	1.30e-06	9.60e-08	3.78e-08
Average Additive Error	5.8e-07	1.1e-07	7.8e-08	1.9e-07	2.7e-09	2.2e-09
Max Additive Error	4.5e-06	1.3e-06	6.9e-07	1.9e-06	2.1e-08	1.8e-08
Average Relative Error:	0.11	0.039	0.08	0.12	0.028	0.062
Max Relative Error	0.41	0.22	0.47	0.65	0.23	0.26
Precision	1	1	1	1	0.99	1
Recall	0.86	0.97	0.9	0.82	0.97	0.9
F_1 Score	0.92	0.99	0.95	0.9	0.98	0.95

It turns out however that *using the frontier is critical* to get good accuracy. This is because nodes in the target set could have high inverse-PPR, which increases the variance. This intuition is confirmed by the scatterplot of the true vs estimated value, shown in Figure 5(d).

• **Balancing Forward and Reverse Work:** Balanced FAST-PPR, as described in Section 3.1, chooses forward and reverse accuracy parameters ϵ_r and ϵ_f dynamically for each target node. Figure 3 shows that Balanced FAST-PPR improves the average running-time across all graphs.

In Figure 6, we plot the forward and reverse running-times for FAST-PPR and Balanced FAST-PPR as a function of the target PageRank. Note that for high global-PageRank targets, FAST-PPR does too much reverse work, while for low global-PageRank targets, FAST-PPR does too much forward work – this is corrected by Balanced FAST-PPR.

Acknowledgements

Peter Lofgren is supported by a National Defense Science and Engineering Graduate (NDSEG) Fellowship.

Ashish Goel and Siddhartha Banerjee were supported in part by the DARPA XDATA program, by the DARPA GRAPHS program via grant FA9550- 12-1-0411 from the U.S. Air Force Office of Scientific Research (AFOSR) and the Defense Advanced Research Projects Agency (DARPA), and by NSF Award 0915040.

C. Seshadhri is at Sandia National Laboratories, which is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

7. REFERENCES

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

[2] G. Jeh and J. Widom, "Scaling personalized web search," in *Proceedings of the 12th international conference on World Wide Web*, ACM, 2003.

[3] P. Yin, W.-C. Lee, and K. C. Lee, "On top-k social web search," in *Proceedings of the 19th ACM international conference on Information and knowledge management*, ACM, 2010.

[4] S. A. Yahia, M. Benedikt, L. V. Lakshmanan, and J. Stoyanovich, "Efficient network aware search in collaborative tagging sites," *Proceedings of the VLDB Endowment*, 2008.

[5] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. d. C. Reis, and B. Ribeiro-Neto, "Efficient search ranking in social networks," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, ACM, 2007.

[6] D. Horowitz and S. D. Kamvar, "The anatomy of a large-scale social search engine," in *Proceedings of the 19th international conference on World wide web*, ACM, 2010.

[7] L. Backstrom and J. Leskovec, "Supervised random walks: predicting and recommending links in social networks," in *Proceedings of the fourth ACM international conference on Web search and data mining*, ACM, 2011.

[8] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *Proceedings of the 22nd international conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2013.

[9] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, 2006.

[10] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, p. 3, ACM, 2012.

[11] H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," 2006.

[12] T. Sarlós, A. A. Benczúr, K. Csalogány, D. Fogaras, and B. Rácz, "To randomize or not to randomize: space optimal summaries for hyperlink analysis," in *Proceedings of the 15th international conference on World Wide Web*, ACM, 2006.

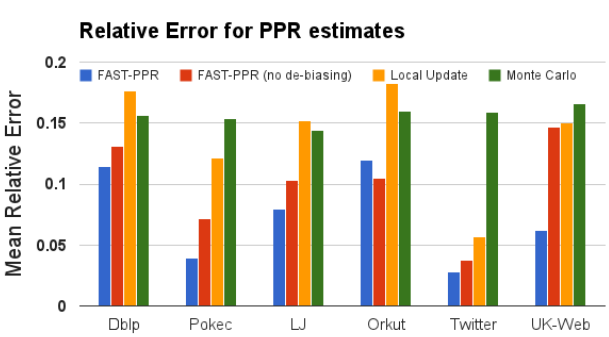
[13] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova, "Monte carlo methods in pagerank computation: When one iteration is sufficient," *SIAM Journal on Numerical Analysis*, 2007.

[14] 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*, Springer, 2007.

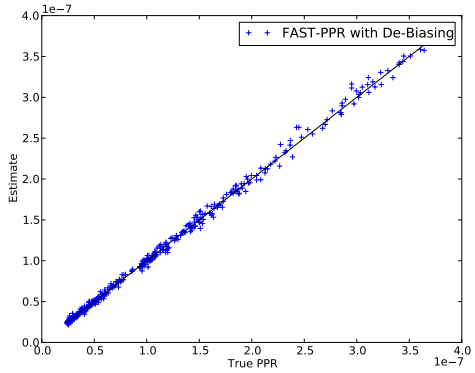
[15] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized pagerank," *Proceedings of the VLDB Endowment*, 2010.

[16] C. Borgs, M. Brautbar, J. Chayes, and S.-H. Teng, "Multi-scale matrix sampling and sublinear-time pagerank computation," *Internet Mathematics*, 2013.

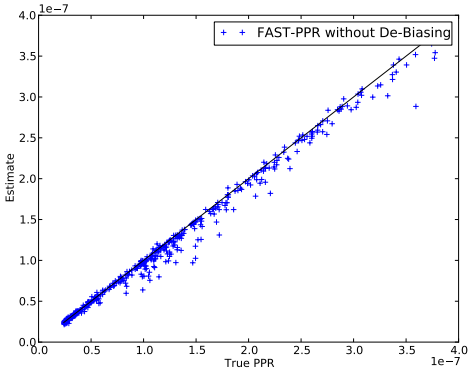
[17] A. D. Sarma, A. R. Molla, G. Pandurangan, and



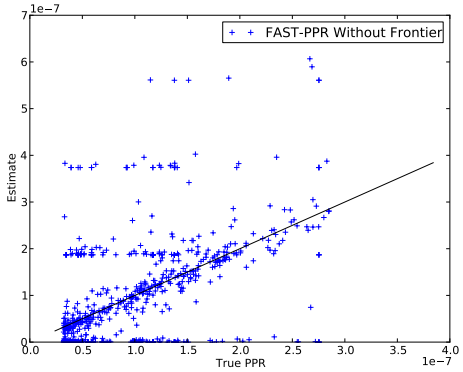
(a) Accuracy of FAST-PPR vs. benchmarks



(b) FAST-PPR with de-biasing



(c) FAST-PPR without de-biasing



(d) FAST-PPR using target set instead of frontier

Figure 5: Accuracy experiments. Figure 5(a) compares FAST-PPR to benchmarks across data sets. Comparing Figure 5(a) to Figure 5(c) shows the value of de-biasing, while 5(d) shows the necessity of the frontier. All experiments conducted on the Twitter 2010 graph.

E. Upfal, "Fast distributed pagerank computation," *Distributed Computing and Networking*, 2013.

[18] "Laboratory for web algorithmics." http://law.di.unimi.it/datasets.php. Accessed: 2014-02-11.

[19] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós, "Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments," *Internet Mathematics*, 2005.

[20] P. Lofgren and A. Goel, "Personalized pagerank to a target node," *arXiv preprint arXiv:1304.4658*, 2013.

[21] O. Goldreich and D. Ron, "On testing expansion in bounded-degree graphs," in *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, Springer, 2011.

[22] S. Kale, Y. Peres, and C. Seshadhri, "Noise tolerance of expanders and sublinear expander reconstruction," in *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, IEEE, 2008.

[23] R. Motwani and P. Raghavan, *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[24] O. Goldreich and D. Ron, "Property testing in bounded degreegraphs," *Algorithmica*, 2002. Conference version in STOC 1997.

[25] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, ACM Press, 2011.

[26] P. Boldi, M. Santini, and S. Vigna, "A large time-aware graph," *SIGIR Forum*, vol. 42, no. 2, 2008.

[27] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International. Scientific Conf. & Workshop Present Day Trends of Innovations*, 2012.

[28] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, (San Diego, CA), October 2007.

[29] "Stanford network analysis platform (snap)." http://http://snap.stanford.edu/. Accessed: 2014-02-11.

APPENDIX

A. FAST-PPR WITH APPROXIMATE FRONTIER ESTIMATES

We now extend to the case where we are given an estimate $\{\widehat{\pi}_t^{-1}(w)\}$ of the inverse-PPR vector of v, with maximum additive error ϵ_{inv}, i.e.:

$$\max_{w \in V} \left\{ |\pi_t^{-1}(w) - \widehat{\pi}_t^{-1}(w)| \right\} < \epsilon_{inv}.$$

We note here that the estimate we obtain from the APPROX-FRONTIER algorithm (Algorithm 2) has a *one-sided* error; it always underestimates π_t^{-1}. However, we prove the result for the more general setting with two-sided error.

First, we have the following lemma:

LEMMA 1. *Given ϵ_r and $\epsilon_{inv} \leq \epsilon_r \beta$ for some $\beta \in (0, 1/2)$. Defining approximate target-set $\widehat{T}_t(\epsilon_r) = \{w \in V : \widehat{\pi}_t^{-1}(w) > \epsilon_r\}$, we have:*

$$T_t((1+\beta)\epsilon_r) \subseteq \widehat{T}_t(\epsilon_r) \subseteq T_t(\epsilon_r(1-\beta))$$

Moreover, let $c_{inv} = \left(\frac{\beta}{1-\beta}\right) < 1$; then $\forall\, w \in \widehat{T}_t(\epsilon_r)$:

$$|\pi_t^{-1}(w) - \widehat{\pi}_t^{-1}(w)| < c_{inv} \pi_t^{-1}(w).$$

PROOF. The first claim follows immediately from our definition of T_t and the additive error guarantee. Next, from the fact that $\widehat{T}_t(\epsilon_r) \subseteq T_t(\epsilon_r(1-\beta))$. we have that $\forall\, w \in \widehat{T}_t(\epsilon_r), \pi_t^{-1}(w) \geq (1-\beta)\epsilon_r$. Combining this with the additive guarantee, we get the above estimate. □

Now for FAST-PPR estimation with approximate oracles, the main problem is that if $\epsilon_{inv} = \beta\epsilon_r$, then the additive error guarantee is not useful when applied to nodes in the frontier: for example, we could have $\pi_t^{-1}(w) < \epsilon_{inv}$ for all w in the frontier $\widehat{F}_t(\epsilon_r)$. On the other hand, if we allow the walks to reach the target set, then the variance of our estimate may be too high, since $\pi_t^{-1}(w)$ could be $\Omega(1)$ for nodes in the target set. To see this visually, compare figures 5(c) and 5(d) – in the first we use the estimate for nodes in the frontier, ending up with a negative bias; in the second, we allow walks to hit the target set, leading to a high variance.

However, Lemma 1 shows that we have a multiplicative guarantee for nodes in $\widehat{T}_t(\epsilon_r)$. We now circumvent the above problems by *re-sampling walks to ensure they do not enter the target set.* We first need to introduce some additional notation. Given a target set T, for any node $u \notin T$, we can partition its neighboring nodes $\mathcal{N}^{out}(u)$ into two sets – $\mathcal{N}_T(u) \triangleq \mathcal{N}^{out}(u) \cap T$ and $\mathcal{N}_{\overline{T}}(u) = \mathcal{N}^{out}(u) \setminus T$. We also define a *target-avoiding random-walk* $RW_T(u) = \{u, V_1, V_2, V_3, \ldots\}$ to be one which starts at u, and at each node w goes to a uniform random node in $\mathcal{N}_{\overline{T}}(w)$.

Now given target-set T, suppose we define the *score* of any node $u \notin T$ as $S_T(u) = \frac{\sum_{z \in \mathcal{N}_T(u)} \pi_z(t)}{d^{out}(u)}$. We now have the following lemma:

LEMMA 2. *Let $RW_T(s) = \{u, V_1, V_2, V_3, \ldots\}$ be a target-avoiding random-walk, and $L \sim Geom(\alpha)$. Further, for any node $u \notin T$, let $p_{\overline{T}}(u) = \frac{\mathcal{N}_{\overline{T}}(u)}{d^{out}(u)}$, i.e., the fraction of neighboring nodes of u not in target set T. Then:*

$$\pi_s(t) = \mathbb{E}_{RW_T(s),L}\left[\sum_{i=1}^{L}\left(\prod_{j=0}^{i-1} p_{\overline{T}}(V_j)\right) S_T(V_i)\right]. \quad (5)$$

Algorithm 4 Theoretical-FAST-PPR

Inputs: start node s, target node t, threshold δ, reverse threshold ϵ_r, relative error c, failure probability p_{fail}.

1: Define $\epsilon_f = \frac{\delta}{\epsilon_r}$, $\beta = \frac{c}{3+c}$, $p_{min} = \frac{c}{3(1+\beta)}$
2: Call FRONTIER(t, ϵ_r) (with β as above) to obtain sets $\widehat{T}_t(\epsilon_r), \widehat{F}_t(\epsilon_r)$, inverse-PPR values $(\widehat{\pi}_t^{-1}(w))_{w \in T_t(\epsilon_r)}$ and target-set entering probabilities $p_{\overline{T}}(u)\, \forall\, u \in \widehat{F}_t(\epsilon_r)$.
3: Set $l_{max} = \log_{1-\alpha}\left(\frac{c\delta}{3}\right)$ and $n_f = \frac{45 l_{max}}{c^2 \epsilon_f} \log\left(\frac{2}{p_{fail}}\right)$
4: **for** index $i \in [n_f]$ **do**
5: Generate $L_i \sim Geom(\alpha)$
6: Setting $T = \widehat{T}_t(\epsilon_r)$, generate target-avoiding random-walk $RW_T^i(s)$ via rejection-sampling of neighboring nodes.
7: Stop the walk $RW_T^i(s)$ when it encounters any node u with $p_{\overline{T}}(u) < p_{min}$, OR, when the number of steps is equal to $\min\{L_i, l_{max}\}$.
8: Compute walk-score S_i as the weighted score of nodes on the walk, according to equation 5.
9: **end for**
10: **return** $\frac{1}{n_f}\sum_{i=1}^{n_f} S_i$

PROOF. Given set T, and any node $u \notin T$, we can write:

$$\pi_u(t) = \frac{1-\alpha}{d^{out}(u)}\left(\sum_{z \in \mathcal{N}_T(u)} \pi_z(t) + \sum_{z \in \mathcal{N}_{\overline{T}}(u)} \pi_z(t)\right).$$

Recall we define the score of node u given T as $S_T(u) = \frac{\sum_{z \in \mathcal{N}_T(u)} \pi_z(t)}{d^{out}(u)}$, and the fraction of neighboring nodes of u not in T as $p_{\overline{T}}(u) = \frac{\mathcal{N}_{\overline{T}}(u)}{d^{out}(u)}$. Suppose we now define the *residue* of node u as $R_T(u) = \frac{\sum_{z \in \mathcal{N}_{\overline{T}}(u)} \pi_z(t)}{|\mathcal{N}_{\overline{T}}(u)|}$. Then we can rewrite the above formula as:

$$\pi_u(t) = (1-\alpha)\left(S_T(u) + p_{\overline{T}}(u) R_T(u)\right).$$

Further, from the above definition of the residue, observe that $R_T(u)$ is the *expected value of $\pi_W(t)$ for a uniformly picked node $W \in \mathcal{N}_{\overline{T}}(u)$*, i.e., a uniformly chosen neighbor of u which is not in T. Recall further that we define a target-avoiding random-walk $RW_T(s) = \{s, V_1, V_2, V_3, \ldots\}$ as one which at each node picks a uniform neighbor not in T. Thus, by iteratively expanding the residual, we have:

$$\begin{aligned}\pi_s(t) &= \mathbb{E}_{RW_T(s)}\left[\sum_{i=1}^{\infty}(1-\alpha)^i\left(\prod_{j=0}^{i-1} p_{\overline{T}}(V_j)\right) S_T(V_i)\right] \\ &= \mathbb{E}_{RW_T(s),L}\left[\sum_{i=1}^{L}\left(\prod_{j=0}^{i-1} p_{\overline{T}}(V_j)\right) S_T(V_i)\right].\end{aligned}$$

□

We can now use the above lemma to get a modified FAST-PPR algorithm, as follows: First, given target t, we find an approximate target-set $T = \widehat{T}(\epsilon_r)$. Next, from source s, we generate a target-avoiding random walk, and calculate the weighted sum of scores of nodes. The walk collects the entire score the first time it hits a node in the frontier; subsequently, for each node in the frontier that it visits, it accumulates a smaller score due to the factor $p_{\overline{T}}$. Note that $p_{\overline{T}}(u)$ is also the probability that a random neighbor

of u is not in the target set. Thus, at any node u, we need to sample on average $1/p_{\overline{T}}(u)$ neighbors until we find one which is not in T. This gives us an efficient way to generate a target-avoiding random-walk using rejection sampling – we stop the random-walk any time the current node u has $p_{\overline{T}}(u) < p_{min}$ for some chosen constant p_{min}, else we sample neighbors of u uniformly at random till we find one not in T. The complete algorithm is given in Algorithm 4.

Before stating our main results, we briefly summarize the relevant parameters. Theoretical-FAST-PPR takes as input a pair of node (s,t), a threshold δ, desired relative error c and desired probability of error p_{fail}. In addition, it involves six internal parameters – $\epsilon_r, \epsilon_f, \beta, p_{min}, l_{max}$ and n_f – which are set based on the input parameters (δ, p_{fail}, c) (as specified in Algorithm 4). ϵ_r and β determine the accuracy of the PPR estimates from FRONTIER; n_f is the number of random-walks; l_{max} and p_{min} help control the target-avoiding walks.

We now have the following guarantee of correctness of Algorithm 4. For ease of exposition, we state the result in terms of relative error of the estimate – however it can be adapted to a classification error estimate in a straightforward manner.

THEOREM 5. *Given nodes (s,t) such that $\pi_s(t) > \delta$, desired failure probability p_{fail} and desired tolerance $c < 1$. Suppose we choose parameters as specified in Algorithm 4. Then with probability at least $1 - p_{fail}$, the estimate returned by Theoretical-FAST-PPR satisfies:*

$$\left| \pi_s(t) - \frac{1}{n_f} \sum_{i=1}^{n_f} S_i \right| < c\pi_s(t).$$

Moreover, we also have the following estimate for the running-time:

THEOREM 6. *Given threshold δ, teleport probability α, desired relative-error c and desired failure probability p_f, the Theoretical-FAST-PPR algorithm with parameters chosen as in Algorithm 4, requires:*

- *Average reverse-time* $= O\left(\frac{d}{\alpha\beta\epsilon_r}\right) = O\left(\frac{d}{c\alpha\epsilon_r}\right)$
- *Average forward-time* $\leq O\left(\frac{n_f}{\alpha p_{min}}\right)$
$= O\left(\frac{\log(1/p_f)\log(1/\delta)}{c^3\alpha\epsilon_f \log(1/(1-\alpha))}\right),$

where $d = \frac{m}{n}$ is the average in-degree of the network.

We note that in Theorem 6, the reverse-time bound is averaged over random (s,t) pairs, while the forward-time bound is averaged over the randomness in the Theoretical-FAST-PPR algorithm (in particular, for generating the target-avoiding random-walks). As before, we can get worst-case runtime bounds by storing the frontier estimates for all nodes. Also, similar to Corollary 1, we can balance forward and reverse times aas follows:

COROLLARY 2. *Let $\epsilon_r = \sqrt{\frac{\delta c^2 d \log(1/(1-\alpha))}{\log(1/p_{fail})\log(1/\delta)}}$, where $d = m/n$, and $\epsilon_f = \delta/\epsilon_r$. Then Theoretical-FAST-PPR has an average running-time of $O\left(\frac{1}{\alpha c^2}\sqrt{\frac{d}{\delta}}\sqrt{\frac{\log(1/p_{fail})\log(1/\delta)}{\log(1/(1-\alpha))}}\right)$ per-query, or alternately, worst-case running time and per-node pre-computation and storage of $O\left(\frac{1}{\alpha c^2}\sqrt{\frac{d}{\delta}}\sqrt{\frac{\log(1/p_{fail})\log(1/\delta)}{\log(1/(1-\alpha))}}\right)$.*

PROOF OF THEOREM 6. The reverse-time bound is same as for Theorem 1. For the forward-time, observe that each target-avoiding random-walk $RW_T(s)$ takes *less than* $1/\alpha = O(1)$ steps on average (since the walk can be stopped before L, either when it hits l_{max} steps, or encounters a node u with $p_{\overline{T}}(u) < p_{min}$). Further, for each node on the walk, we need (on average) at most $1/p_{min}$ samples to discover a neighboring node $\notin T$. □

PROOF OF THEOREM 5. Define target set $T = \widehat{T}_t(\epsilon_r)$. Using Lemma 2, given a target-avoiding random-walk $RW_T(s) = \{s, V_1, V_2, \ldots\}$, we define our estimator for $\pi_s(t)$ as:

$$\widehat{\pi}_s(t) = \sum_{i=1}^{L} \left(\prod_{j=0}^{i-1} p_{\overline{T}}(V_j) \right) \widehat{S}_T(V_i).$$

Now, from Lemma 1, we have that for any node $u \notin T$, the *approximate score* $\widehat{S}_T(u) = \frac{1}{d^{out}(u)} \sum_{z \in \mathcal{N}_T(u)} \widehat{\pi}_z(t)$ lies in $(1 \pm c_{inv}) S_T(u)$. Thus, for any $s \notin \widehat{T}_t(\epsilon_r)$, we have:

$$|\mathbb{E}_{RW_T}[\widehat{\pi}_s(t)] - \pi_s(t)| \leq c_{inv}\pi_s(t)$$

with $c_{inv} = \frac{\beta}{1-\beta} = \frac{c}{3}$ (as we chose $\beta = \frac{c}{3+c}$).

Next, suppose we define our estimate as $S = \frac{1}{n_f}\sum_{i=1}^{n_f} S_i$. Then, from the triangle inequality we have:

$$|S - \pi_s(t)| \leq |S - \mathbb{E}[S]| + |\mathbb{E}[S] - \mathbb{E}[\widehat{\pi}_s(t)]| + \quad (6)$$
$$|\mathbb{E}_{RW_T}[\widehat{\pi}_s(t)] - \pi_s(t)|. \quad (7)$$

We have already shown that the third term is bounded by $c\pi_s(t)/3$. The second error term is caused due to two mutually exclusive events – stopping the walk due to truncation, or due to the current node having $p_{\overline{T}}(u)$ less than p_{min}. To bound the first, we can re-write our estimate as:

$$\widehat{\pi}_u(t) = \sum_{i=1}^{L} \left(\prod_{j=0}^{i-1} p_{\overline{T}}(V_j) \right) \widehat{S}_T(V_i)$$
$$\leq \mathbb{E}_{RW_T}\left[\sum_{i=1}^{L \wedge l_{max}} \left(\prod_{j=0}^{i-1} p_{\overline{T}}(V_j) \right) S_T(V_i) \right] + (1-\alpha)^{(l_{max}+1)},$$

where $L \sim Geom(\alpha)$, and $L \wedge l_{max} = \min\{L, l_{max}\}$ for any $l_{max} > 0$. Choosing $l_{max} = \log_{1-\alpha}(c\delta/3)$, we incur an additive loss in truncation which is at most $c\delta/3$ – thus for any pair (s,t) such that $\pi_s(t) > \delta$, the loss is at most $c\pi_s(t)/3$. On the other hand, if we stop the walk at any node u such that $p_{\overline{T}}(u) \leq p_{min}$, then we lose at most a p_{min} fraction of the walk-score. Again, if $\pi_s(t) > \delta$, then we have from before that $\widehat{\pi}_s(t) < \delta(1+\beta)$ – choosing $p_{min} = \frac{c}{3(1+\beta)}$, we again get an additive loss of at most $c\pi_s(t)/3$.

Finally, to show a concentration for S, we need an upper bound on the estimates S_i. For this, note first that for any node u, we have $\widehat{S}_T(u) \leq \epsilon_r$. Since we truncate all walks at length l_{max}, the per-walk estimates S_i lie in $[0, l_{max}\epsilon_r]$. Suppose we define $T_i = \frac{S_i}{l_{max}\epsilon_r}$ and $T = \sum_{i \in [n_f]} T_i$; then we have $T_i \in [0,1]$. Also, from the first two terms in equation 6, we have that $|\mathbb{E}[S_i] - \pi_s(t)| < 2c\pi_s(t)/3$, and thus $\left(1 - \frac{2c}{3}\right)\frac{n_f \pi_s(t)}{l_{max}\epsilon_r} \leq \mathbb{E}[T] \leq \left(1 + \frac{2c}{3}\right)\frac{n_f \pi_s(t)}{l_{max}\epsilon_r}$. Now, as in The-

orem 2, we can now apply standard Chernoff bounds to get:

$$\begin{aligned}
\mathbb{P}\left[|S-\mathbb{E}[S]| \geq \frac{c\pi_s(t)}{3}\right] &= \mathbb{P}\left[|T-\mathbb{E}[T]| \geq \frac{cn_f\pi_s(t)}{3l_{max}\epsilon_r}\right] \\
&\leq \mathbb{P}\left[|T-\mathbb{E}[T]| \geq \frac{c\mathbb{E}[T]}{3(1+2c/3)}\right] \\
&\leq 2\exp\left(-\frac{\left(\frac{c}{3+2c}\right)^2\mathbb{E}[T]}{3}\right) \\
&\leq 2\exp\left(-\frac{c^2(1-2c/3)n_f\pi_s(t)}{3(3+2c)^2 l_{max}\epsilon_r}\right) \\
&\leq 2\exp\left(-\frac{c^2(3-2c)n_f\epsilon_f}{9(3+2c)^2 l_{max}}\right).
\end{aligned}$$

Since $c \in [0,1]$, setting $n_f = \frac{45}{c^2}\cdot\frac{l_{max}\log(2/p_{fail})}{\epsilon_f}$ gives the desired failure probability. □

B. LOWER BOUND FOR SIGNIFICANT-PPR QUERIES

Proof of Theorem 3. We perform a direct reduction. Set $N = \lfloor 1/10\delta \rfloor$, and consider the distributions $\mathcal{G}_1$ and $\mathcal{G}_2$. We will construct a distinguisher using a single query to an algorithm for *Detect-High*(δ). Hence, if there is an algorithm for *Detect-High*(δ) taking less than $1/100\sqrt{\delta}$ queries, then Theorem 4 will be violated. We construct a distinguisher as follows. Given graph G, it picks two uniform random nodes s and t. We run the algorithm for *Detect-High*(δ). If it accepts (so it declares $\pi_s(t) > \delta$), then the distinguisher outputs $\mathcal{G}_1$ as the distribution that G came from. Otherwise, it outputs $\mathcal{G}_2$.

We prove that the distinguisher is correct with probability $> 2/3$. First, some preliminary lemmas.

Lemma 3. *With probability $> 3/4$ over the choice of G from $\mathcal{G}_1$, for any nodes s,t, $\pi_s(t) > \delta$.*

Proof. A graph formed by picking 3 uniform random matchings is an *expander* with probability $1-\exp(-\Omega(N)) > 3/4$ (Theorem 5.6 of [23]). Suppose G was an expander. Then a random walk of length $10\log N$ from s in G is guaranteed to converge to the uniform distribution. Formally, let W be the random walk matrix of G. For any $\ell \geq \lfloor 10\log N \rfloor$, $\|W^\ell \mathbf{e}_s - \mathbf{1}/N\|_2 \leq 1/N^2$, where $\mathbf{1}$ is the all ones vector [23]. So for any such ℓ, $(W^\ell \mathbf{e}_s)(t) \geq 1/2N$. By standard expansions, $\pi_s = \sum_{\ell=0}^{\infty}\alpha(1-\alpha)^\ell W^\ell \mathbf{e}_s$. We can bound

$$\begin{aligned}
\pi_s(t) &\geq \sum_{\ell \geq \lfloor 10\log N\rfloor} \alpha(1-\alpha)^\ell (W^\ell \mathbf{e}_s)(t) \\
&\geq (2N)^{-1}\sum_{\ell \geq \lfloor 10\log N\rfloor}\alpha(1-\alpha)^\ell \\
&= (1-\alpha)^{\lfloor 10\log N\rfloor}(2N)^{-1}
\end{aligned}$$

Setting $\alpha = 1/100\log(1/\delta)$ and $N = \lfloor 1/10\delta \rfloor$, we get $\pi_s(t) > 1/6N \geq \delta$. All in all, when G is an expander, $\pi_s(t) > \delta$. □

Lemma 4. *Fix any G from $\mathcal{G}_2$. For two uniform random nodes s,t in G, $\pi_s(t) = 0$ with probability at least $3/4$.*

Proof. Any graph in G has 4 connected components, each of size $N/4$. The probability that s and t lie in different components is $3/4$, in which case $\pi_s(t) = 0$. □

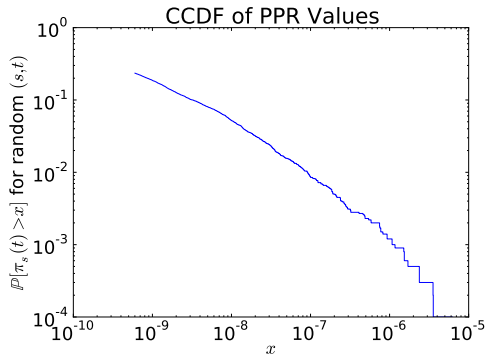


Figure 7: Complementary cumulative distribution for 10,000 (s,t) pairs sampled uniformly at random on the Twitter graph.

If the adversary chose $\mathcal{G}_1$, then $\pi_s(t) > \delta$. If he chose $\mathcal{G}_2$, $\pi_s(t) = 0$ – by the above lemmas, each occurs with probability $> 3/4$. In either case, the probability that *Detect-High*(δ) errs is at most $1/10$. By the union bound, the distinguisher is correct overall with probability at least $2/3$.

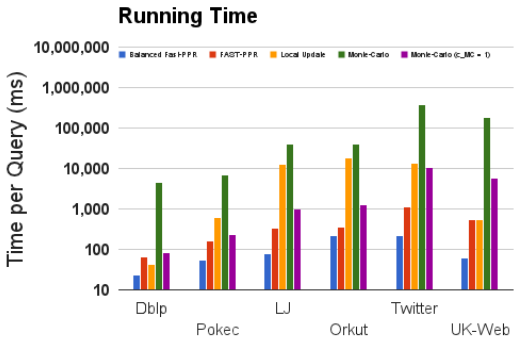
C. SOME ADDITIONAL PLOTS

C.1 Distribution of PPR values

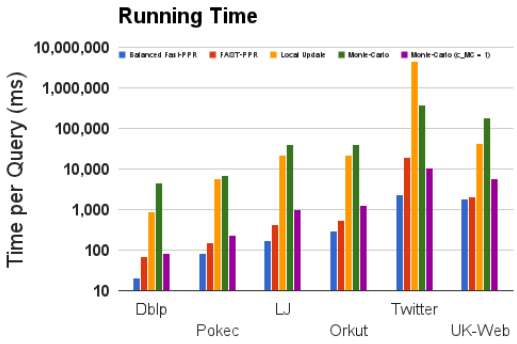
For our running-time experiments, we use $\delta = \frac{4}{n}$. To understand how informative this threshold is, we study the distribution of PPR values. Using the Twitter graph as an example, we chose 10,000 random (s,t) pairs and compute $\pi_s(t)$ using FAST-PPR to accuracy $\delta = \frac{n}{10}$. The complementary cumulative distribution function is shown on a log-log plot in Figure 7. Notice that the plot is roughly linear, suggesting a power-law. Because of this skewed distribution, only 2.8% of pairs have PPR PPR above $\frac{1}{n}$ = 2.4e-8, and less than 1% have value over $\frac{4}{n}$ = 9.6e-8.

C.2 Runtime vs. less-Accurate Monte-Carlo

In Figure 8 we show the data data as in Figure 3 with the addition of a less accurate version of Monte-Carlo. When using Monte-Carlo to detect an event with probability ϵ_f, the minimum number of walks needed to see the event once on average is $\frac{1}{\epsilon_f}$. The average relative error of this method is significantly greater than the average relative error of FAST-PPR, but notice that even with this minimal number of walks, Balanced FAST-PPR is faster than Monte-Carlo.



(a) Sampling targets uniformly



(b) Sampling targets from PageRank distribution

Figure 8: Average running-time (on log-scale) for different networks, with $\delta = \frac{4}{n}, \alpha = 0.2$, for 1000 (s,t) pairs. We compare to a less-accurate version of Monte-Carlo, with only $\frac{1}{\epsilon_f}$ walks – notice that Balanced FAST-PPR is still faster than Monte-Carlo.