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 Management Science & Engineering
Stanford University
sidb@stanford.edu

Ashish Goel
Department of Management Science & Engineering
Stanford University
ashishg@stanford.edu

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

ABSTRACT

We propose a new algorithm, FAST-PPR, for estimating personalized PageRank: given start node s and target node t in a directed graph, and given a threshold δ, FAST-PPR estimates the Personalized PageRank $\pi_s(t)$ from s to t, guaranteeing a small relative error as long $\pi_s(t) > \delta$. Existing algorithms for this problem have a running-time of $\Omega(1/\delta)$; in comparison, FAST-PPR has a provable average running-time guarantee of $O(\sqrt{d/\delta})$ (where d is the average in-degree of the graph). This is a significant improvement, since δ is often $O(1/n)$ (where n is the number of nodes) for applications. We also complement the algorithm with an $\Omega(1/\sqrt{\delta})$ lower bound for PageRank estimation, 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, on the 2010 Twitter graph with 1.5 billion edges, for target nodes sampled by popularity, 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.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on Discrete Structures
; G.2.2 [**Graph Theory**]: Graph Algorithms

General Terms

Algorithms,Theory

KDD '14, August 24 - 27 2014, New York, NY, USA.

ACM 978-1-4503-2956-9/14/08 ...$15.00.
http://dx.doi.org/10.1145/2623330.2623745.

Keywords

Personalized PageRank; Social Search

1. INTRODUCTION

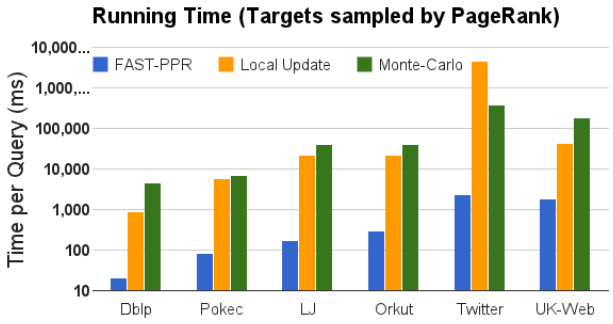
The success of modern networks is largely due to the ability to search effectively on them. A key primitive is PageRank [1], which is widely used as a measure of network importance. The popularity of PageRank is in large part due to its fast computation in large networks. As modern social network applications shift towards being more customized to individuals, there is a need for similar 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. In a typical personalized search application, given a set of candidate results for a query, we want to estimate the Personalized PageRank to each candidate result. This motivates the following problem:

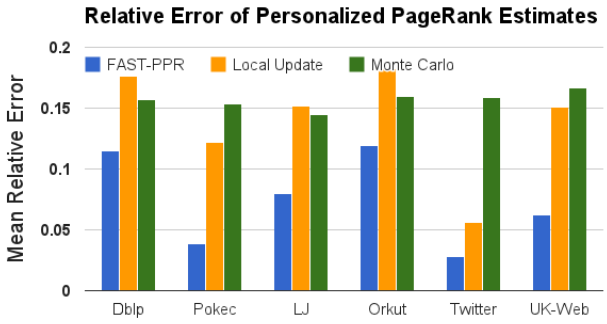
Given source node s and target node t, estimate the Personalized PageRank $\pi_s(t)$ up to a small relative error.

Since smaller values of $\pi_s(t)$ are more difficult to detect, we parameterize the problem by threshold δ, requiring small relative errors only if $\pi_s(t) > \delta$. Current techniques used for PPR estimation (see Section 2.1) have $\Omega(1/\delta)$ running-time – this makes them infeasible for large networks when the desired $\delta = O(1/n)$ 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,



(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.

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), based on a new bi-directional search technique for PPR estimation:

• **Practical Contributions:** We present a simple implementation of FAST-PPR which requires no pre-processing and has an average running-time of $O(\sqrt{d/\delta})$[1]. We also propose a simple heuristic, Balanced FAST-PPR, that achieves a significant speedup in practice.

In experiments, FAST-PPR 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 variety of data sets, FAST-PPR is significantly faster than the state of the art, with the same or better accuracy.

To give a concrete example: in experiments on the Twitter-2010 graph [18], Balanced FAST-PPR takes less than 3 seconds 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 20 times faster than the state-of-the-art, without sacrificing accuracy.

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

We also give a new running-time *lower bound* of $\Omega(1/\sqrt{\delta})$ for PPR estimation, 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], where the source is picked from a distribution over nodes. Different preference vectors are used for various applications [2, 1]. However, for simplicity of presentation, we focus on Personalized PageRank in this work.

2. PRELIMINARIES

Given $G(V, E)$, a directed graph, with $|V| = n, |E| = m$, and adjacency matrix A. For any node $u \in V$, we denote $\mathcal{N}^{out}(u), d^{out}(u)$ as the out-neighborhood and out-degree respectively; similarly $\mathcal{N}^{in}(u), d^{in}(u)$ are the in-neighborhood and 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 the following random walk starting from u: at each step, return to u with probability α, and otherwise move 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 for practical applications, we need to consider δ of the form $O(1/n)$ or $O(\log n/n)$. We reinforce this choice of δ using empirical PPR data from the Twitter-2010 graph in Section 6.2 – in particular, we observe that only 1

2.1 Existing Approaches for PPR Estimation

There are two main techniques used to compute PageRank/PPR vectors. One set of algorithms use 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, similar to the Jacobi

[1]We assume here that the desired relative error and teleport probability α are constants – complete scaling details are provided in our Theorem statements in Appendix A.

[2]Formally, for (s,t) with $\pi_s(t) > \delta$, FAST-PPR returns an estimate $\widehat{\pi}_s(t)$ with relative error c, incurring $O\left(\frac{1}{c^2}\sqrt{\frac{d}{\delta}}\sqrt{\frac{\log(1/p_{fail})\log(1/\delta)}{\alpha^2\log(1/(1-\alpha))}}\right)$ average running-time see Corollary 2 in Appendix A for details.

iteration. This technique was first proposed by Jeh and Widom [2], and subsequently improved by other researchers [19, 9]. The 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'). However, one can exhibit graphs where these algorithms need a running-time of $O(1/\delta)$ to get additive guarantees on the order of δ.

Eqn. 2 can be derived from the following probabilistic re-interpretations of PPR, which also lead to an alternate set of randomized or *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 random length $L \sim Geom(\alpha)$[3], starting from u. Then we can write:

$$\pi_u(v) = \mathbb{P}[V_L = v] \tag{3}$$

In other words, $\pi_u(v)$ is the probability that v is the last node in $RW(u, L)$. Another alternative characterization is:

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

i.e., $\pi_u(v)$ is proportional to the number of times $RW(u, L)$ visits node v. Both characterizations 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 estimating PPR values close to the desired 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 a PPR-estimation query with parameters (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.

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

Our method is most easily understood using the characterization of $\pi_s(t)$ as the probability that a single walk from s ends at t (Eqn. (3)). Consider a random walk from s to t – at some point, it must enter the frontier. We can then decompose the probability of the walk reaching t into the product of two probabilities: the probability that it reaches some 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. Figure 2 illustrates this bi-directional scheme.

2.3 Additional Definitions

To formalize our algorithm, we first need some additional definitions. We define a set B to be a blanket set for t with respect to s if all paths 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 (defining $H_B = \perp$ if the walk does not hit B before terminating). Since each walk corresponds to a unique H_B, we can write $\mathbb{P}[V_L = v]$ as a sum of contributions from each node in B. Further, from the memoryless property of the geometric random variable, the probability a walk ends at t conditioned on reaching $w \in B$ before stopping is exactly $\pi_s(t)$. Combining these, we have:

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

In other words, the PPR from s to t is the sum, over all nodes w in blanket set B, of the probability of a random walk hitting B first 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 introduce two crucial definitions:

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:*

$$F_t(\epsilon_r) := \Big(\bigcup_{v \in T_t(\epsilon_r)} \mathcal{N}_v^{in} \Big) \setminus T_t(\epsilon_r).$$

The target set $T_t(\epsilon_r)$ thus 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 $T_t(\epsilon_r)$.

2.4 A Bidirectional Estimator

The next proposition illustrates the core of our approach.

PROPOSITION 1. *Set $\epsilon_r < \alpha$. Fix vertices s, t such that $s \notin T_t(\epsilon_r)$.*

1. *Frontier set $F_t(\epsilon_r)$ is a blanket set of t with respect to s.*
2. *For random-walk $RW(s, L)$ with length $L \sim Geom(\alpha)$:*

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

PROOF. For brevity, let $T_t = T_t(\epsilon_r), F_t = F_t(\epsilon_r)$. By definition, we know $\pi_t(t) \geq \alpha$ – thus $t \in T_t$, since $\epsilon_r < \alpha$. The frontier set F_t contains all neighbors of nodes in T_t which are not themselves in T_t – hence, for any source node $u \notin T_t$, a path to t must first hit a node in F_t.

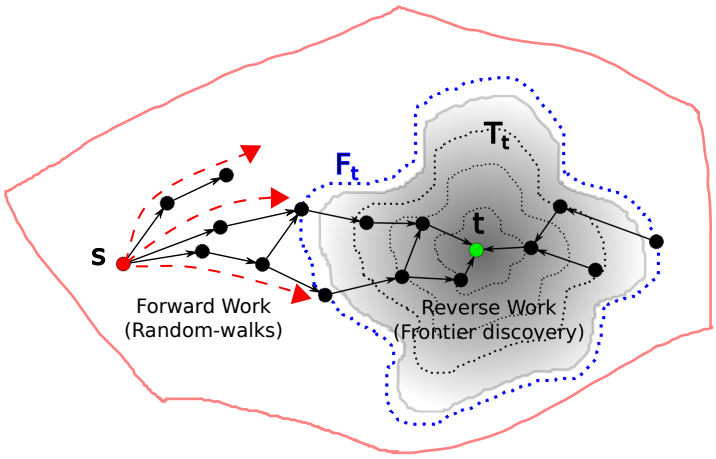


Figure 2: The FAST-PPR Algorithm: We first work backward from the target t to find the frontier F_t (with inverse-PPR estimates). Next we work forward from the source s, generating random-walks and testing for hitting the frontier.

For the second part, since F_t is a blanket set for s with respect to t, Eqn. 4 implies $\pi_s(t) = \sum_{w \in F_t} \mathbb{P}[H_{F_t} = w]\, \pi_w(t)$, where H_{F_t} is the first node where the walk hits F_t. Note that by definition, $\forall\, w \in F_t$ we have $w \notin T_t$ – thus $\pi_w(t) \leq \epsilon_r$. Applying this bound, we get:

$$\pi_s(t) \leq \epsilon_r \sum_{w \in F_t} \mathbb{P}[H_{F_t} = w] = \epsilon_r \mathbb{P}[RW(s, L) \text{ hits } F_t(\epsilon_r)].$$

Rearranging, we get the result. □

The aim is to estimate $\pi_s(t)$ through Eqn. 4. By the previous proposition, $F_t(\epsilon_r)$ is a blanket set. We will determine the set $F_t(\epsilon_r)$ and estimate all quantities in the right side of Eqn. 4, thereby estimating $\pi_s(t)$.

We perform a simple heuristic calculation to argue that setting $\epsilon_r \approx \sqrt{\delta}$ suffices to estimate $\pi_s(t)$. Previous work shows that $T_t(\epsilon_r)$ can be found in $O(d/\epsilon_r)$ time [14, 20] – using this we can find $F_t(\epsilon_r)$. Now suppose that we know all values of $\pi_w(t)$ $(\forall\, w \in F_t(\epsilon_r))$. By Eqn. 4, we need to estimate the probability of random walks from s hitting vertices in $F_t(\epsilon_r)$. By the previous proposition, the probability of hitting $F_t(\epsilon_r)$ is at least δ/ϵ_r – hence, we need $O(\epsilon_r/\delta)$ walks from s to ensure we hit $F_t(\epsilon_r)$. All in all, we require $O(d/\epsilon_r + \epsilon_r/\delta)$ – setting $\epsilon_r = \sqrt{d\delta}$ we get a running-time bound of $O(\sqrt{d/\delta})$. In reality, however, we only have (coarse) PPR estimates for nodes in the frontier – we show how these estimates can be boosted to get the desired guarantees, and also empirically show that, in practice, using the frontier estimates gives good results. Finally, we show that $1/\sqrt{\delta}$ is a fundamental lower bound for this problem.

We note that bi-directional techniques have been used for estimating fixed-length random walk probabilities in *regular undirected graphs* [21, 22].

These techniques do not extend to estimating PPR – in particular, we need to consider *directed graphs, arbitrary node degrees and walks of random length.* Also, Jeh and Widom [2] proposed a scheme for PageRank estimation using intermediate estimates from a *fixed skeleton* of target nodes. However there are no running-time guarantees for such schemes; also, the target nodes and partial estimates need to be pre-computed and stored. Our algorithm is fundamentally different as it constructs *separate target sets for each target node* at query-time.

3. THE FAST-PPR ALGORITHM

We now develop the *Frontier-Aided Significance Thresholding* algorithm, or FAST-PPR, specified in Algorithm 1. The input-output behavior of FAST-PPR is as follows:

• **Inputs:** The primary inputs are graph G, teleport probability α, start node s, target node t, and threshold δ – for brevity, we henceforth suppress the dependence on G and α. We also need a *reverse threshold* ϵ_r – in subsequent sections, we discuss how this parameter is chosen.

• **Output:** An estimate $\widehat{\pi}_s(t)$ for $\pi_s(t)$.

The algorithm also requires two parameters, c and β – the former controls the number of random walks, while the latter controls the quality of our inverse-PPR estimates in the target set. In our pseudocode (Algorithms 1 and 2), we specify the values we use in our experiments – the theoretical basis for these choices is provided in Section 3.2.

Algorithm 1 FAST-PPR(s, t, δ)

Inputs: Graph G, teleport probability α, start node s, target node t, threshold δ

Set accuracy parameters c, β (in our experiments we use $c = 350$, $\beta = 1/6$).
Call FRONTIER(t, ϵ_r, β) to obtain target set $T_t(\epsilon_r)$, frontier set $F_t(\epsilon_r)$, and inverse-PPR values $(\pi_t^{-1}(w))_{w \in F_t(\epsilon_r) \cup T_t(\epsilon_r)}$.
if $s \in T_t(\epsilon_r)$ **then**
  return $\pi_t^{-1}(s)$
else
  Set number of walks $k = c\epsilon_r/\delta$ (cf. Theorem 3)
  for index $i \in [k]$ **do**
    Generate $L_i \sim Geom(\alpha)$
    Generate random-walk $RW(s, L_i)$
    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 $H_i = \perp$
  end for
  return $\widehat{\pi}_s(t) = (1/k) \sum_{i \in [k]} \pi_t^{-1}(H_i)$
end if

Algorithm 2 FRONTIER(t, ϵ_r, β) [14, 20]

Inputs: Graph G, teleport probability α, target node t, reverse threshold ϵ_r, accuracy factor β

Define additive error $\epsilon_{inv} = \beta\epsilon_r$
Initialize (sparse) estimate-vector $\widehat{\pi}_t^{-1}$ and (sparse) residual-vector r_t as: $\begin{cases} \widehat{\pi}_t^{-1}(u) = r_t(u) = 0 \text{ if } u \neq t \\ \widehat{\pi}_t^{-1}(t) = r_t(t) = \alpha \end{cases}$
Initialize target-set $\widehat{T}_t = \{t\}$, frontier-set $\widehat{F}_t = \{\}$
while $\exists w \in V\ s.t.\ r_t(w) > \alpha\epsilon_{inv}$ **do**
  for $u \in \mathcal{N}^{in}(w)$ **do**
    $\Delta = (1 - \alpha) \cdot \frac{r_t(w)}{d^{out}(u)}$
    $\widehat{\pi}_t^{-1}(u) = \widehat{\pi}_t^{-1}(u) + \Delta, r_t(u) = r_t(u) + \Delta$
    if $\widehat{\pi}_t^{-1}(u) > \epsilon_r$ **then**
      $\widehat{T}_t = \widehat{T}_t \cup \{u\}$, $\widehat{F}_t = \widehat{F}_t \cup \mathcal{N}^{in}(u)$
    end if
  end for
  Update $r_t(w) = 0$
end while
$\widehat{F}_t = \widehat{F}_t \setminus \widehat{T}_t$
return $\widehat{T}_t, \widehat{F}_t, (\widehat{\pi}_t^{-1}(w))_{w \in \widehat{F}_t \cup \widehat{T}_t}$

FAST-PPR needs to know sets $T_t(\epsilon_r), F_t(\epsilon_r)$ and inverse-PPR values $(\pi_t^{-1}(w))_{w \in F_t(\epsilon_r) \cup T_t(\epsilon_r)}$. These can be obtained (approximately) from existing algorithms of [14, 20]. For the sake of completeness, we provide pseudocode for the procedure FRONTIER (Algorithm 2) that obtains this information. The following combines Theorems 1 and 2 in [20].

THEOREM 1. *FRONTIER(t, ϵ_r, β) algorithm computes estimates $\widehat{\pi}_t^{-1}(w)$ for every vertex w, with a guarantee that $\forall w$, $|\widehat{\pi}_t^{-1}(w) - \pi_t^{-1}(w)| < \beta\epsilon_r$. The average running time (over all choices of t) is $O(d/(\alpha\epsilon_r))$, where $d = m/n$ is the average degree of the graph.*

Observe that the estimates $\widehat{\pi}_t^{-1}(w)$ are used to find approximate target and frontier sets. Note that the running time for a given t is proportional to the frontier size $|\widehat{F}_t|$. It is relatively straightforward to argue (as in [20]) that $\sum_{t \in V} |\widehat{F}_t| = \Theta(nd/(\alpha\epsilon_r))$.

In the subsequent subsections, we present theoretical analyses of the running times and correctness of FAST-PPR. The correctness proof makes an excessively strong assumption of perfect outputs for FRONTIER, which is not true. To handle this problem, we have a more complex variant of FAST-PPR that can be proven theoretically (see Appendix A). Nonetheless, our empirical results show that FAST-PPR does an excellent job of estimate $\pi_s(t)$.

3.1 Running-time of FAST-PPR

THEOREM 2. *Given parameters δ, ϵ_r, the running-time of the FAST-PPR algorithm, averaged over uniform-random pairs s, t, is $O(\alpha^{-1}(d/\epsilon_r + \epsilon_r/\delta))$.*

PROOF. Each random walk $RW(u, Geom(\alpha))$ takes $1/\alpha$ steps on average, and there are $O(\epsilon_r/\delta)$ such walks performed – this is the *forward time*. On the other hand, from Theorem 1, we have that for a random (s, t) pair, the average running time of FRONTIER is $O(d/(\alpha\epsilon_r))$ – this is the *reverse time*. Combining the two, we get the result. □

Note that the reverse time bound above is averaged across choice of target node; for some target nodes (those with high global PageRank) the reverse time may be much larger than average, while for others it may be smaller. However, the forward time is similar for all source nodes, and is predictable – we exploit this in Section 5.2 to design a balanced version of FAST-PPR which is much faster in practice. In terms of theoretical bounds, the above result suggests an obvious choice of ϵ_r to optimize the running time:

COROLLARY 1. *Set $\epsilon_r = \sqrt{d\delta}$. Then FAST-PPR has an average per-query running-time of $O(\alpha^{-1}\sqrt{d/\delta})$.*

3.2 FAST-PPR with Perfect FRONTIER

We now analyze FAST-PPR in an idealized setting, where we assume that FRONTIER returns *exact inverse-PPR estimates* – i.e., the sets $T_t(\epsilon_r)$, $F_t(\epsilon_r)$, and the values $\{\pi_t^{-1}(w)\}$ are known exactly. This is an unrealistic assumption, but it gives much intuition into *why* FAST-PPR works. In particular, we show that if $\pi(s, t) > \delta$, then with probability at least 99

THEOREM 3. *For any s, t, δ, ϵ_r, FAST-PPR outputs an estimate $\widehat{\pi}_s(t)$ such that with probability > 0.99:*

$$|\pi_s(t) - \widehat{\pi}_s(t)| \leq \max(\delta, \pi_s(t))/4.$$

PROOF. We choose $c = \max(48 \cdot 8e \ln(100), 4\log_2(100))$; this choice of parameter c is for ease of exposition in the computations below, and has not been optimized.

To prove the result, note that FAST-PPR performs $k = c\epsilon_r/\delta$ i.i.d. random walks $RW(s, L)$. We use H_i to denote the first node in $F_t(\epsilon_r)$ hit by the ith random walk. Let $X_i = \pi_t^{-1}(H_i)$ and $X = \sum_{i=1}^{k} X_i$. By Eqn. 4, $\mathbb{E}[X_i] = \pi_s(t)$, so $\mathbb{E}[X] = k\pi_s(t)$. Note that $\widehat{\pi}_s(t) = X/k$. As result, $|\pi_s(t) - \widehat{\pi}_s(t)|$ is exactly $|X - \mathbb{E}[X]|/k$.

It is convenient to define scaled random variables $Y_i = X_i/\epsilon_r$ and $Y = X/\epsilon_r$ before we apply standard Chernoff bounds. We have $|\pi_s(t) - \widehat{\pi}_s(t)| = (\epsilon_r/k)|Y - \mathbb{E}[Y]| = (\delta/c)|Y - \mathbb{E}[Y]|$. Also, $\pi_s(t) = (\delta/c)\mathbb{E}[Y]$. Crucially, because $H_i \in F_t(\epsilon_r)$, $X_i = \pi_{H_i}(t) < \epsilon_r$, so $Y_i \leq 1$. Hence, we can apply the following two Chernoff bounds (refer to Theorem 1.1 in [23]):

1. $\mathbb{P}[|Y - \mathbb{E}[Y]| > \mathbb{E}[Y]/4] < \exp(-\mathbb{E}[Y]/48)$
2. For any $b > 2e\mathbb{E}[Y], \mathbb{P}[Y > b] \leq 2^{-b}$

Now, we perform a case analysis. Suppose $\pi_s(t) > \delta/(4e)$. Then $\mathbb{E}[Y] > c/(4e)$, and

$$\begin{aligned} \mathbb{P}[|\pi_s(t) - \widehat{\pi}_s(t)| > \pi_s(t)/4] &= \mathbb{P}[|Y - \mathbb{E}[Y]| > \mathbb{E}[Y]/4] \\ &< \exp(-c/48 \cdot 4e) < 0.01 \end{aligned}$$

Suppose $\pi_s(t) \leq \delta/(4e)$. Then, $\delta/4 > 2e\pi_s(t)$ implying $c/4 > 2e\mathbb{E}[Y]$. By the upper tail:

$$\mathbb{P}[\widehat{\pi}_s(t) > \delta/4] = \mathbb{P}[Y > c/4] \leq 2^{-c/4} < 0.01$$

The proof is completed by trivially combining both cases. □

4. LOWER BOUND FOR PPR ESTIMATION

In this section, we prove that any algorithm that accurately estimates PPR queries up to a threshold δ must look at $\Omega(1/\sqrt{\delta})$ edges of the graph. Thus, our algorithms have the optimal dependence on δ. The numerical constants below 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 4. *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 equally sized 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 5 (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 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 5. The complete proof is provided in Appendix B. □

5. FURTHER VARIANTS OF FAST-PPR

The previous section describes vanilla FAST-PPR, with a proof of correctness assuming a perfect FRONTIER. We now present some variants of FAST-PPR. We give a theoretical variant that is a truly provable algorithm, with no assumptions required – however, vanilla FAST-PPR is a much better practical candidate and it is what we implement. We also discuss how we can use pre-computation and storage to obtain worst-case guarantees for FAST-PPR. Finally, from the practical side, we discuss a workload-balancing heuristic that provides significant improvements in running-time by dynamically adjusting ϵ_r.

5.1 Using Approximate Frontier-Estimates

The assumption that FRONTIER returns perfect estimates is theoretically untenable – Theorem 1 only ensures that each inverse-PPR estimate is correct up to an additive factor of $\epsilon_{inv} = \beta\epsilon_r$. It is plausible that for every w in the frontier $\widehat{F}_t(\epsilon_r)$, $\pi_t^{-1}(w) < \epsilon_{inv}$, and FRONTIER may return a zero estimate for these PPR values. It is not clear how to use these noisy inverse-PPR estimates to get the desired accuracy guarantees.

To circumvent this problem, we observe that estimates $\pi_t^{-1}(w)$ for any node $w \in \widehat{T}_t(\epsilon_r)$ *are in fact accurate up to a multiplicative factor*. We design a procedure to bootstrap these 'good' estimates, by using a special 'target-avoiding' random walk – this modified algorithm gives the desired accuracy guarantee with only an additional $\log(1/\delta)$ factor in running-time. The final algorithm and proof are quite intricate – due to lack of space, the details are deferred to Appendix A.

5.2 Balanced FAST-PPR

In FAST-PPR, the parameter ϵ_r can be chosen freely while preserving accuracy. Choosing a larger value leads to a smaller frontier and less forward work at the cost of more reverse work; a smaller value requires more reverse work and fewer random walks. To improve performance in practice, we can optimize the choice of ϵ_r based on the target t, to balance the reverse and forward time. Note that for any value of ϵ_r, the forward time is proportional to $k = c\epsilon_r/\delta$ (the number of random walks performed) – for any choice of ϵ_r, it is easy to estimate the forward time required. Thus, instead of committing to a single value of ϵ_r, we propose a heuristic, BALANCED-FAST-PPR, wherein we *dynamically decrease ϵ_r until the estimated remaining forward time equals the reverse time already spent.*

We now describe BALANCED-FAST-PPR in brief: Instead of pushing from any node w with residual $r_t(w)$ above a fixed threshold $\alpha\epsilon_{inv}$, we now push from the node w with the largest residual value – this follows a similar algorithm proposed in [20]. From [14, 20], we know that a current maximum residual value of r_{max} implies an additive error guarantee of $\frac{r_{max}}{\alpha}$ – this corresponds to a dynamic ϵ_r value of $\frac{r_{max}}{\alpha\beta}$. At this value of ϵ_r, the number of forward walks required is $k = c\epsilon_r/\delta$. By multiplying k by the average time needed to generate a walk, we get a good prediction of the amount of forward work still needed – we can then compare it to the time already spent on reverse-work and adjust ϵ_r until they are equal. Thus BALANCED-FAST-PPR is able to dynamically choose ϵ_r to balance the forward and reverse running-time. Complete pseudocode is in Appendix C. In Section 6.5, we experimentally show how this change balances forward and reverse running-time, and significantly reduces the average running-time.

5.3 FAST-PPR using Stored Oracles

All our results for FAST-PPR have involved average-case running-time bounds. To convert these to worst-case running-time bounds, we can pre-compute and store the frontier for all nodes, and only perform random walks at query time. To obtain the corresponding storage requirement for these *Frontier oracles*, 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:

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

To further cut down on running-time, we can also pre-compute and store the random-walks from all nodes, and perform appropriate joins at query time to get the FAST-PPR estimate. This allows us to implement FAST-PPR on any distributed system that can do fast intersections/joins. More generally, it demonstrates how the modularity of FAST-PPR can be used to get variants that trade-off between different resources in practical implementations.

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: use of frontier and balancing forward/reverse running-time?

6.1 Experimental Setup

Table 1: Datasets used in experiments

Dataset	Type	# Nodes	# Edges
DBLP-2011	undirected	1.0M	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

• **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://cs.stanford.edu/~plofgren/fast_ppr/

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

Monte-Carlo refers to 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 D).

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 $\delta/2$. Note that a backward local-update is more suited to computing PPR forward schemes [9, 2] as the latter lack natural performance guarantees on graphs with high-degree nodes.

• **Parameters:** For FAST-PPR, we set the constants $c = 350$ and $\beta = 1/6$ – these are guided by the Chernoff bounds we use in the proof of Theorem 3. For vanilla FAST-PPR, we simply choose $\epsilon_r = \sqrt{\delta}$.

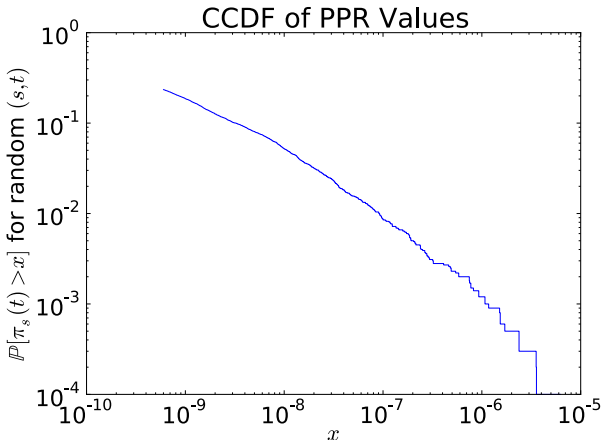


Figure 3: Complementary cumulative distribution for 10,000 (s,t) pairs sampled uniformly at random on the Twitter graph.

6.2 Distribution of PPR values

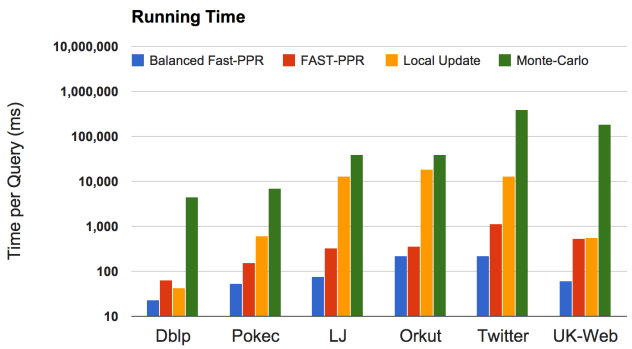
For all our experiments, we use $\delta = \frac{4}{n}$. To understand the importance of this threshold, we study the distribution of PPR values in real networks. Using the Twitter graph as an example, we choose 10,000 random (s,t) pairs and compute $\pi_s(t)$ using FAST-PPR to accuracy $\delta = \frac{n}{10}$. The complementary cumulative distribution function is shown on a log-log plot in Figure 3. Notice that the plot is roughly linear, suggesting a power-law. Because of this skewed distribution, only 2.8

6.3 Running Time Comparisons

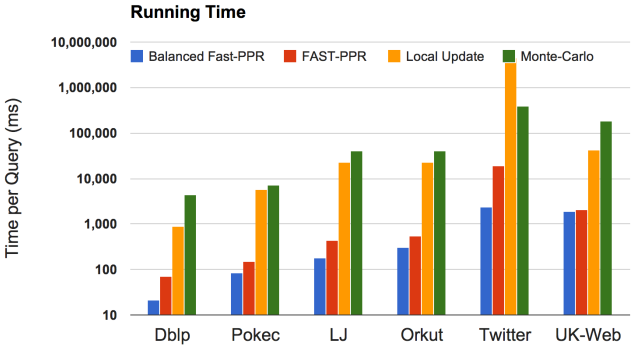
After loading the graph into memory, we sample 1000 source/target pairs (s,t) uniformly at random. For each, we measure the time required for answering PPR-estimation queries with threshold $\delta = 4/n$, which, as we discuss above, is fairly significant because of the skew of the PPR distribution. 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 4 – we compare Monte-Carlo, Local-Update, vanilla FAST-PPR, and BALANCED-FAST-PPR. We perform an analogous experiment where target nodes are sampled according to their global PageRank value. This is a more realistic model for queries in personalized search applications, with searches biased towards more popular targets. The results, plotted in Figure 4(b), show even better speedups for FAST-PPR. All in all, FAST-PPR is many orders of magnitude faster than the state of the art.

The effect of target global PageRank: To quantify the speedup further, we sort the targets in the Twitter-2010 graph by their global PageRank, and choose the first target in each percentile. We measure the running-time of the four algorithms (averaging over random source nodes), as shown in Figure 5. 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



(a) Sampling targets uniformly



(b) Sampling targets from PageRank distribution

Figure 4: 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 4(a), or from the global PageRank distribution in Figure 4(b). In this plot we use teleport probability $\alpha = 0.2$.

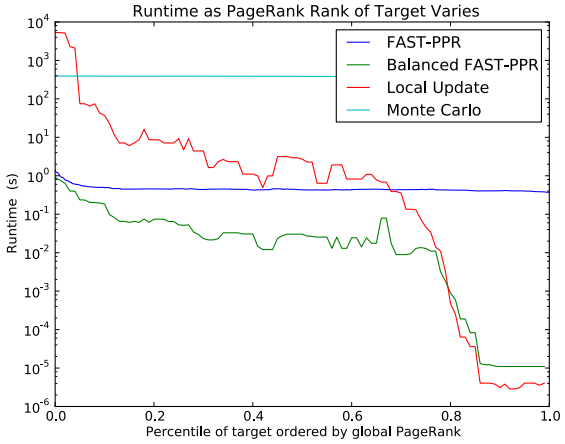


Figure 5: Execution time vs. global PageRank of the target. Target nodes were sorted by global PageRank, then one was chosen from each percentile. We use $\alpha = 0.2$, and 5-point median smoothing.

the average running-time – thus improving performance in these cases causes significant speedups in the average running time. We also see that BALANCED-FAST-PPR has significant improvements over vanilla FAST-PPR, especially for lower PageRank targets.

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 those from Local Update, and 6 times more accurate than those from Monte Carlo.

6.4 Measuring the Accuracy of FAST-PPR

We measure the empirical accuracy of FAST-PPR. For each graph, we sample 25 targets uniformly at random, and compute their ground truth inverse-PPR vectors by running a power iteration up to an additive error of $\delta/100$ (as before, we use $\delta = 4/n$). Since larger PPR values are easier to compute than smaller PPR values, we sample start nodes such that $\pi_s(t)$ is near the significance threshold δ. In particular, for each of the 25 targets t, we sample 50 random nodes from the set $\{s : \delta/4 \leq \pi_s(t) \leq \delta\}$ and 50 random 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. Notice that FAST-PPR has mean relative error less than 15

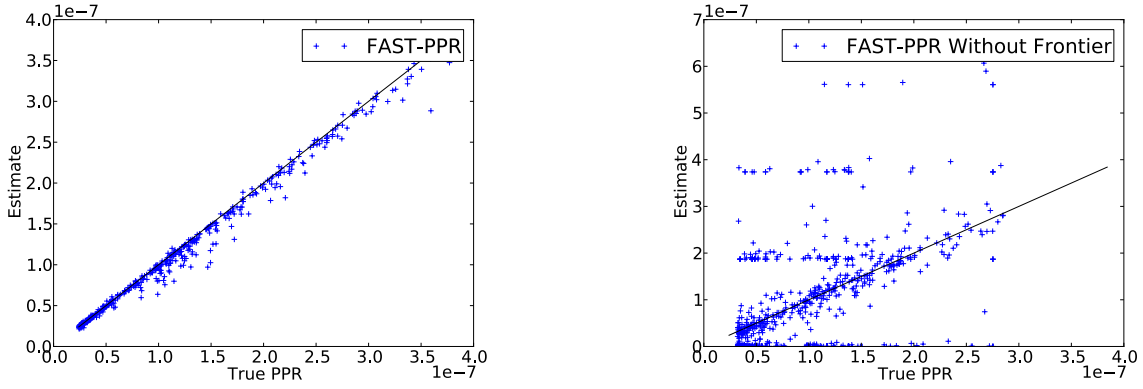
To make sure that FAST-PPR is not sacrificing accuracy for improved running-time, we also compute the relative error of Local-Update and Monte-Carlo, using the same parameters as for our running-time experiments. For each of the 2500 (s, t) pairs, we run Local-Update, and measure its relative error. For testing Monte-Carlo, we use our knowledge of the ground truth PPR, and the fact that each random-walk from s terminates at t with probability $p_s = \pi_s(t)$. This allows us to simulate Monte-Carlo by directly sampling from a Bernoulli variable with mean $\pi_s(t)$ – this statistically identical to generating random-walks and testing over all pairs. Note that actually simulating the walks would take more than 50 days of computation for 2500 pairs. The relative errors are shown in Figure 1(b). Notice that FAST-PPR is more accurate than the state-of-the-art competition on all graphs. This shows that our running time comparisons are using parameters settings that are fair.

6.5 Some Other Empirical Observations

• **Necessity of the Frontier:** Another question we study experimentally 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 the target set is also a blanket set, and we have good approximations for inverse-PPR values in the target set. Further, using only the target set would reduce the dependence of the running-time

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



(a) FAST-PPR

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

Figure 6: The importance of using the frontier. In each of these plots, a perfect algorithm would place all data points on the line $y = x$. Notice how using inverse-PPR estimates from the target set rather than the frontier results in significantly worse accuracy.

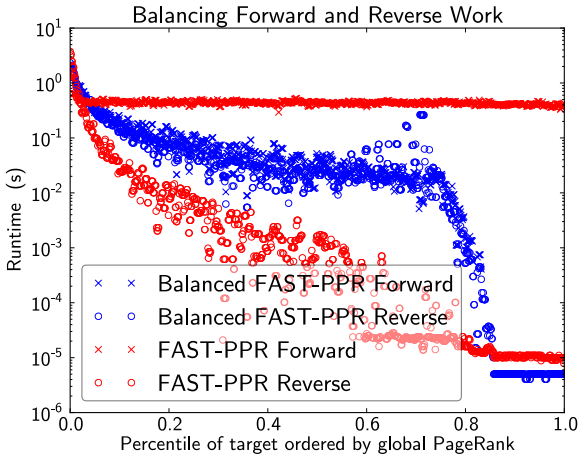


Figure 7: Forward and reverse running-times (on log-scale) for FAST-PPR (in red) and BALANCED-FAST-PPR (in blue) as we vary the global PageRank of the target node on the x-axis. Data is for the Twitter-2010 graph and is smoothed using median-of-five smoothing. Notice how there is a significant gap between the forward and backward work in FAST-PPR, and that this gap is corrected by BALANCED-FAST-PPR.

on d, and also reduce storage requirements for an oracle-based implementation.

It turns out however that *using the frontier is critical* to get good accuracy. Intuitively, this is because nodes in the target set may have high inverse-PPR, which then increases the variance of our estimate. This increase in variance can be visually seen in a scatterplot of the true vs estimated value, as shown in Figure 6(b) – note that the estimates generated using the frontier set are much more tightly clustered around the true PPR values, as compared to the estimates generated using the target set.

• **Balancing Forward and Reverse Work:** BALANCED-FAST-PPR, as described in Section 5.2, chooses reverse threshold ϵ_r dynamically for each target node. Figure 4 shows that BALANCED-FAST-PPR improves the average running-time across all graphs.

In Figure 7, 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 De-

fense 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 socialnetworks," in *Proceedings of the fourth ACM international conference on Web searchand 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 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] D. Dubhashi and A. Panconesi, *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009.

[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.

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

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 6(a) and 6(b) – 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:*

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

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

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

Algorithm 3 Theoretical-FAST-PPR(s, t, δ)

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** $\frac{1}{n_f}\sum_{i=1}^{n_f} S_i$

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 3.

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 3). 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 3. 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 6. *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 3. 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 7. *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 3, 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 7, 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 7. The reverse-time bound is same as for Theorem 2. 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 6. 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 3, 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. DETAILS OF LOWER BOUND

Proof of Theorem 4. 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 5 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 [30]). 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 [30]. 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 $\pi_s(t) = 0$. □

Algorithm 4 BALANCED-FAST-PPR(s,t,δ)

Inputs: Graph G, teleport probability α, start node s, target node t, threshold δ

1: Set accuracy parameters c, β (These trade-off speed and accuracy – in experiments we use $c = 350$, $\beta = 1/6$.)
2: Call BALANCED-FRONTIER(G,α,t,c,β) to obtain target set $T_t(\epsilon_r)$, frontier set $F_t(\epsilon_r)$, inverse-PPR values $(\pi_t^{-1}(w))_{w\in F_t(\epsilon_r)\cup T_t(\epsilon_r)}$, and reverse threshold ϵ_r
3: (The rest of BALANCED-FAST-PPR is identical to FAST-PPR (Algorithm 1) from line 3)

Algorithm 5 BALANCED-FRONTIER(t,c,β)

Inputs: Graph $G=(V,E)$, teleport probability α, target node t, accuracy factors c,β.

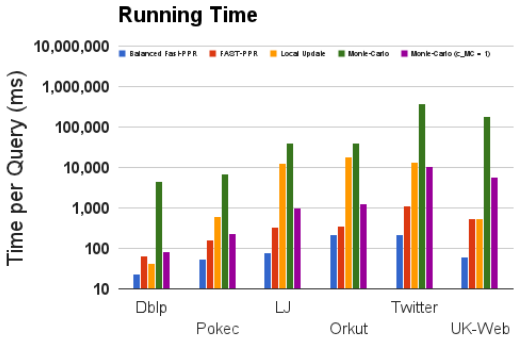
1: Set constant T_{walk} to be the (empirical) average time it takes to generate a walk.
2: Initialize (sparse) estimate-vector $\widehat{\pi}_t^{-1}$ and (sparse) residual-vector r_t as: $\begin{cases}\widehat{\pi}_t^{-1}(u) = r_t(u) = 0 \text{ if } u \neq t \\ \widehat{\pi}_t^{-1}(t) = r_t(t) = \alpha\end{cases}$
3: Initialize target-set $\widehat{T}_t = \{t\}$, frontier-set $\widehat{F}_t = \{\}$
4: Initialize $\epsilon_r = 1/\beta$.
5: Start a timer to track time spent.
6: Define function FORWARD-TIME(ϵ_r) $= T_{walk}\cdot k$ where $k = c\cdot\epsilon_r/\delta$ is the number of walks needed.
7: **while** time spent $<$ FORWARD-TIME(ϵ_r) **do**
8: $w = \arg\max_u r_t(u)$
9: **for** $u \in \mathcal{N}^{in}(w)$ **do**
10: $\Delta = (1-\alpha).\frac{r_t(w)}{d^{out}(u)}$
11: $\widehat{\pi}_t^{-1}(u) = \widehat{\pi}_t^{-1}(u) + \Delta, r_t(u) = r_t(u) + \Delta$
12: **if** $\widehat{\pi}_t^{-1}(u) > \epsilon_r$ **then**
13: $\widehat{T}_t = \widehat{T}_t \cup \{u\}$, $\widehat{F}_t = \widehat{F}_t \cup \mathcal{N}^{in}(u)$
14: **end if**
15: **end for**
16: Update $r_t(w) = 0, \epsilon_r = (\max_u r_t(u))/(\alpha\cdot\beta)$
17: **end while**
18: $\widehat{F}_t = \widehat{F}_t \setminus \widehat{T}_t$
19: **return** $\widehat{T}_t, \widehat{F}_t, (\widehat{\pi}_t^{-1}(w))_{w\in\widehat{F}_t\cup\widehat{T}_t}, \epsilon_r$

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 $2/3$.

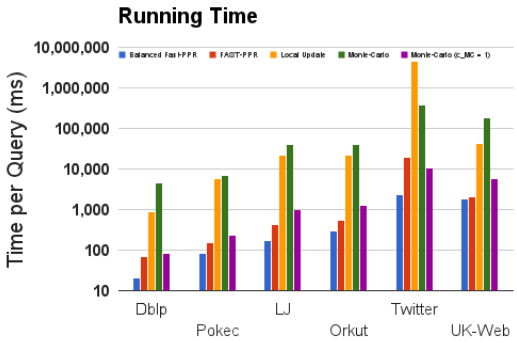
C. BALANCED-FAST-PPR PSEUDOCODE

We introduced BALANCED-FAST-PPR in Section 5.2, where for brevity, we only gave a brief description. We provide the complete pseudocode for the Balanced FAST-PPR algorithm in Algorithms 4 and 5.

We note that similar to Algorithm 2, the update rule for inverse-PPR estimates in Algorithm 5 is based on local-update algorithms from [14, 20] – the novelty here is in using runtime estimates for random-walks to determine the amount of backward work (i.e., up to what accuracy the local-update step should be executed). Also note that in Algorithm 5, we need to track the maximum residual estimate – this can be done efficiently using a binary heap. For more details, please refer our source code, which is available at: http://cs.stanford.edu/~plofgren/fast_ppr/



(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.

D. RUNTIME VS. LESS-ACCURATE MONTE-CARLO

In Figure 8 we show the data data as in Figure 4 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.