FORA: Simple and Effective Approximate Single-Source Personalized PageRank

Sibo Wang*
University of Queensland
sibo.wang@uq.edu.au

Renchi Yang
Nanyang Technological University
rcyang@ntu.edu.sg

Xiaokui Xiao
Nanyang Technological University
xkxiao@ntu.edu.sg

Zhewei Wei†
Renmin University of China
zhewei@ruc.edu.cn

Yin Yang
Hamad Bin Khalifa University
yyang@hbku.edu.qa

ABSTRACT

Given a graph G, a source node s and a target node t, the *personalized PageRank* (*PPR*) of t with respect to s is the probability that a random walk starting from s terminates at t. A *single-source PPR* (*SSPPR*) query enumerates all nodes in G, and returns the top-k nodes with the highest PPR values with respect to a given source node s. SSPPR has important applications in web search and social networks, e.g., in Twitter's Who-To-Follow recommendation service. However, SSPPR computation is immensely expensive, and at the same time resistant to indexing and materialization. So far, existing solutions either use heuristics, which do not guarantee result quality, or rely on the strong computing power of modern data centers, which is costly.

Motivated by this, we propose FORA, a simple and effective index-based solution for approximate SSPPR processing, with rigorous guarantees on result quality. The basic idea of FORA is to combine two existing methods Forward Push (which is fast but does not guarantee quality) and Monte Carlo Random Walk (accurate but slow) in a simple and yet non-trivial way, leading to an algorithm that is both fast and accurate. Further, FORA includes a simple and effective indexing scheme, as well as a module for top-k selection with high pruning power. Extensive experiments demonstrate that FORA is orders of magnitude more efficient than its main competitors. Notably, on a billion-edge Twitter dataset, FORA answers a top-500 approximate SSPPR query within 5 seconds, using a single commodity server.

KEYWORDS

Personalized PageRank, Forward Push, Random Walk

*Work done while at NTU.
†Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
KDD'17, August 13-17, 2017, Halifax, NS, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4887-4/17/08…$15.00
https://doi.org/10.1145/3097983.3098072

1 INTRODUCTION

Personalized PageRank (*PPR*) is a fundamental operation first proposed by Google [19], a major search engine. Specifically, given a graph G and a pair of nodes s, t in G, the PPR value $\pi(s, t)$ is defined as the probability that a random walk starting from s (called the *source node*) terminates at t (the *target node*), which reflects the importance of t with respect to s. One particularly useful variant of PPR is the *single-source PPR* (*SSPPR*), which takes as input a source node s and a parameter k, and returns the top-k nodes in G with the highest PPR values with respect to s. According to a recent paper [12], Twitter, a leading microblogging service, applies SSPPR in their Who-To-Follow application, which recommends to a user s (who is a node in the social graph) a number of other users (with high PPR values with respect to s) that user s might want to follow. Clearly, such an application computes SSPPR for every user in the social graph on a regular basis. Hence, accelerating PPR computation may lead to improved user experience (e.g., faster response time), as well as reduced operating costs (e.g., lower power consumption in the data center).

Similar to PageRank [19], PPR computation on a web-scale graph is immensely expensive, which involves extracting eigenvalues of a $n \times n$ matrix, where n is the number of nodes that can reach millions or even billions in a social graph. Meanwhile, unlike PageRank, PPR values cannot be easily materialized: since each pair of source/target nodes lead to a different PPR value, storing all possible PPR values requires $O(n^2)$ space, which is infeasible for large graphs. For these reasons, much previous work focuses on *approximate PPR* computation (defined in Section 2.1), which provides a controllable tradeoff between the execution time and result accuracy. Meanwhile, compared to heuristic solutions, approximate PPR provides rigorous guarantees on result quality.

However, even under the approximate PPR definition, SSPPR computation remains a challenging problem, since it requires sifting through all nodes in the graph. To our knowledge, the majority of existing methods (e.g., [15, 16, 22]) focus on approximate pairwise (i.e., with given source and target nodes) PPR computations. A naive solution is to compute pair-wise PPR $\pi(s, v)$ for each possible target node v, and subsequently applies top-k selection. Clearly, the running time of this approach grows linearly to the number of nodes in the graph, which is costly for large graphs.

Motivated by this, we propose *FORA* (short for **FO**ward Push and **RA**ndom Walks), an efficient algorithm for approximate SSPPR computation. The basic idea of FORA is to combine two

existing solutions in a simple and yet non-trivial way, which are (i) Forward Push [1], which can either computes the exact SSPPR results at a high cost, or terminate early but with no guarantee at all on the result quality, and (ii) Monte Carlo [8], which samples and executes random walks and provides rigorous guarantees on the accuracy of SSPPR results, but is rather inefficient. In fact, this idea is so effective that even without any indexing, basic FORA already outperforms its main competitors BiPPR [15] and HubPPR [22]. Then, we describe a simple and effective indexing scheme for FORA, as well as a novel algorithm for top-k selection. Extensive experiments using several real graphs demonstrate that FORA is more than two orders of magnitude faster than BiPPR, and more than an order of magnitude faster than HubPPR. In particular, on a billion-edge Twitter graph, FORA answers top-500 SSPPR query within 5 seconds, using a single commodity server.

2 BACKGROUND

2.1 Problem Definition

Let $G = (V, E)$ be a directed graph. In case the input graph is undirected, we simply convert it to a directed one by treating each edge as two directed edges of opposing directions. Given a source node $s \in V$ and a decay factor α, a random walk (or more precisely, random walk with restart [10]) from s is a traversal of G that starts from s and, at each step, either (i) terminates at the current node with α probability, or (ii) proceeds to a randomly selected out-neighbor of the current node. For any node $v \in V$, the personalized PageRank (PPR) $\pi(s, v)$ of v with respect to s is then the probability that a random walk from s terminates at v [19].

A *single-source PPR* (*SSPPR*) query takes as input a graph G, a source node s, and a parameter k, and returns the top-k nodes with the highest PPR values with respect to s, together with their respective PPR values. This paper focuses on approximate SSPPR processing, and we first define a simpler version of the approximate SSPPR without top-k selection (called approximate whole-graph SSPPR), as follows.

Definition 2.1 (Approximate Whole-Graph SSPPR). Given a source node s, a threshold δ, an error bound ϵ, and a failure probability p_f, an approximate whole-graph SSPPR query returns an estimated PPR $\hat{\pi}(s, v)$ for each node $v \in V$, such that for any $\pi(s, v) > \delta$,

$$|\pi(s, v) - \hat{\pi}(s, v)| \leq \epsilon \cdot \pi(s, v) \quad (1)$$

holds with at least $1 - p_f$ probability. □

The above definition is consistent with existing work, e.g., [15, 16, 22]. Next we define the approximate top-k SSPPR, as follows.

Definition 2.2 (Approximate Top-k SSPPR). Given a source node s, a threshold δ, an error bound ϵ, a failure probability p_f, and a positive integer k, an approximate top-k SSPPR query returns a sequences of k nodes, $v_1, v_2, \cdots, v_k$, such that with probability $1 - p_f$, for any $i \in [1, k]$ with $\pi(s, v_i^*) > \delta$,

$$|\hat{\pi}(s, v_i) - \pi(s, v_i)| \leq \epsilon \cdot \pi(s, v_i) \quad (2)$$

$$\pi(s, v_i) \geq (1 - \epsilon) \cdot \pi(s, v_i^*) \quad (3)$$

hold with at least $1 - p_f$ probability, where v_i^* is the node whose actual PPR with respect to s is the i-th largest. □

Notation	Description
G=(V, E)	The input graph G with node set V and edge set E
n, m	The number of nodes and edges in G, respectively
$N^{out}(v)$	The set of out-neighbors of node v
$N^{in}(v)$	The set of in-neighbors of node v
$\pi(s, t)$	The exact PPR value of t with respect to s
α	The probability that a random walk terminates at a step
δ, ϵ, p_f	Parameters of an approximate PPR query, as in Definitions 2.1 and 2.2
r_{max}	The residue threshold for local update
$r(s, v)$	The residue of v during a local update process from s
$\pi^{\circ}(s, v)$	The reserve of v during a local update process from s
r_{sum}	The sum of all nodes' residues during a local update process from s

Table 1: Frequently used notations.

Note that Equation 2 ensures the accuracy of the estimated PPR values, while Equation 3 guarantees that the i-th result returned has a PPR value close to the i-th largest PPR score. This definition is consistent with previous work [22]. Following previous work [15, 16, 22], we assume that $\delta = O(1/n)$, where n is the number of nodes in G. The intuition is that, we provide approximation guarantees for nodes with above-average PPR values.

In addition, most applications of personalized PageRank concern web graphs and social networks, in which case the underlying input graphs are generally *scale-free*. That is, for any $k \geq 1$, the fraction $f(k)$ of nodes in G that have k edges satisfies

$$f(k) = c \cdot k^{-\gamma}, \quad (4)$$

where γ is a parameter with $2 \leq \gamma \leq 3$, and c is a constant smaller than 1. It can be verified that, in a scale-free graph with $2 \leq \gamma \leq 3$, the average node degree $m/n = O(\log n)$. We will analyze the asymptotic performance of our algorithm on both general graphs and scale-free graphs. Table 1 lists the frequently-used notations throughout the paper.

Remark. Our algorithms can also handle SSPPR queries where the source s is not fixed but sampled from a node distribution. Interested readers are referred to Appendix B.2 for details.

2.2 Main Competitors

Monte-Carlo. A classic solution for approximate PPR processing is the *Monte-Carlo (MC)* approach [8]. Given a source node s, MC generates ω random walks from s, and it records, for each node v, the fraction of random walks $f(v)$ that terminate at v. It then uses $f(v)$ as an estimation of the PPR $\hat{\pi}(s, v)$ of v with respect to s. According to [8], MC satisfies Definition 2.1 with a sufficiently large number of random walks: $\omega = \Omega\left(\frac{\log(1/p_f)}{\epsilon^2 \delta}\right)$. According to Refs. [15, 16, 22] as well as our experiments in Section 5, MC is rather inefficient. Specifically, the time complexity of MC is $O\left(\frac{\log(1/p_f)}{\epsilon^2 \delta}\right)$. As will be explained later in Section 3.2, when $\delta = O(1/n)$ and the

Algorithm 1: Forward Push

Input: Graph G, source node s, probability α, residue threshold r_{max}
Output: $\pi^\circ(s, v)$, $r(s, v)$ for all $v \in V$

1 $r(s, s) \leftarrow 1$; $r(s, v) \leftarrow 0$ for all $v \neq s$;
2 $\pi^\circ(s, v) \leftarrow 0$ for all v;
3 **while** $\exists v \in V$ *such that* $r(s, v)/|N^{out}(v)| > r_{max}$ **do**
4 **for** *each* $u \in \mathcal{N}^{out}(v)$ **do**
5 $r(s, u) \leftarrow r(s, u) + (1 - \alpha) \cdot \frac{r(s,v)}{|N^{out}(v)|}$
6 $\pi^\circ(s, v) \leftarrow \pi^\circ(s, v) + \alpha \cdot r(s, v)$;
7 $r(s, v) \leftarrow 0$;

graph is scale-free, in which case $m/n = O(\log n)$, this time complexity is a factor of $1/\epsilon$ larger than that of *FORA* even without indexing or top-k pruning.

BiPPR and HubPPR. *BiPPR* [15] and its successor *HubPPR* [22] are currently the states of the art for answering *pairwise* PPR queries, in which both the source node s and the target node t are given, and the goal is to approximate the PPR value $\pi(s, t)$ of t with respect to s. The main idea of BiPPR is a bi-direction search on the input graph G. The forward direction simply samples and executes random walks, akin to MC described above. Unlike MC, however, BiPPR requires a much smaller number of random walks, thanks to additional information provided by the backward search.

The backward search in BiPPR (dubbed as *reverse push*) is originally proposed in [1], and is rather complicated. In a nutshell, the reverse push starts from the target node t, and recursively propagate *residue* and *reserve* values along the reverse directions of edges in G. Initially, the residue is 1 for node t, and 0 for all other nodes. The original reverse push [1] requires complete propagation until the residues of all nodes become very small, which is rather inefficient as pointed out in [15]. BiPPR performs the same backward propagations, but terminates early when the residues of all nodes are below a pre-defined threshold. Then, the method performs forward search, i.e., random walks, utilizing the residue and reserve information computed during backward search. The main tricky part in BiPPR is how to set this residue threshold to minimize computation costs, while satisfying Inequality 1. Intuitively, if the residue threshold is set too high, then the forward search requires numerous random walks to reach the approximation guarantee; conversely, if the residue threshold is too low, then the cost of backward search dominates. Ref. [15] provides a careful analysis, and reports that a residue threshold of $O\left(\epsilon \cdot \sqrt{\frac{m \cdot \delta}{n \cdot \log(1/p_f)}}\right)$ strikes a good balance between forward and backward searches, and achieves a low overall cost for pair-wise PPR computation.

To extend BiPPR to SSPPR, one simple method is to enumerate all nodes in G, and compute the PPR value for each of them with respect to the source node s. The problem, however, is that the residue threshold designed in [15] is not optimized for SSPPR, leading to poor performance. To explain, observe that applying BiPPR for SSPPR involves one backward search at each node in G, but only one single forward search from s. Therefore, we improve the performance of BiPPR by tuning down overhead of each backward search at the cost of a less efficient forward search. This optimization turns out to be non-trivial, and we present it in Appendix B.1. Nevertheless, the properly optimized version of BiPPR still involves high costs since it either *(i)* degrades to the Monte-Carlo approach if the residue threshold is large or *(ii)* incurs a large number of backward searches if the residue threshold is small.

HubPPR [22] is an index structure based on BiPPR that features an improved algorithm for top-k queries. Since HubPPR inherits the deficiencies of the BiPPR, it is not suitable for SSPPR, either. We will demonstrate this in our experiments in Section 5.

Forward Push. *Forward Push* [2] is an earlier solution that is not as efficient as BiPPR and HubPPR. We describe it in detail here since the proposed solution FORA uses its components. Specifically, Forward Push can compute the *exact* PPR values at a high cost. It can also be configured to terminate early, but without any guarantee on result quality. Algorithm 1 shows the pseudo-code of Forward Push for whole-graph SSPPR processing. It takes as input G, a source node s, a probability value α, and a threshold r_{max}; its output consists of two values for each node v in G: a *reserve* $\pi^\circ(s, v)$ and a *residue* $r(s, v)$. The reserve $\pi^\circ(s, v)$ is an approximation of $\pi(s, v)$, while the residue $r(s, v)$ is a by-product of the algorithm. In the beginning of the algorithm, it sets $r(s, s) = 1$ and $\pi^\circ(s, s) = 0$, and sets $r(s, v) = \pi^\circ(s, v) = 0$ for any $v \neq s$ (Lines 1-2 in Algorithm 1). Subsequently, the residue of s is converted into other nodes' reserves and residues in an iterative process (Lines 3-7).

Specifically, in each iteration, the algorithm first identifies every node v with $\frac{r(s,v)}{|N^{out}(v)|} > r_{max}$, where N^{out} denotes the set of out-neighbors of v (Line 3). After that, it *propagates* part of v's residue to each u of v's out-neighbors, increasing u's residue by $(1 - \alpha) \cdot \frac{r(s,v)}{|N^{out}(v)|}$. Then, it increases v's reserve by $\alpha \cdot r(s, v)$, and resets v's residue to $r(s, v) = 0$. This iterative process terminates when every node v has $\frac{r(s,v)}{|N^{out}(v)|} \leq r_{max}$ (Line 3).

Andersen et al. [2] show that Algorithm 1 runs in $O(1/r_{max})$ time, and that the reserve $\pi^\circ(s, v)$ can be regarded as an estimation of $\pi(s, v)$. This estimation, however, does not offer any worst-case assurance in terms of absolute or relative error. As a consequence, Algorithm 1 itself is insufficient for addressing the problem formulated in Definitions 2.1 and 2.2.

3 FORA

This section presents the proposed FORA algorithm. We first describe a simpler version of FORA for whole-graph SSPPR (Definition 2.1) without indexing in Sections 3.1 and Sections 3.2. Then, we present the indexing scheme of FORA in Section 3.3, and top-k selection in Section 3.4.

3.1 Main Idea

As reviewed in Section 2.2, (i) MC is inefficient due to a large number of random walks required to satisfy the approximation guarantee, (ii) BiPPR and HubPPR either degrade to MC, or require fewer forward random walks but still incur high cost due to numerous backward search operations, and (iii) Forward Push with early termination provides no formal guarantee on result quality. The proposed solution FORA can be understood as a combination of these methods. In particular, FORA first performs Forward Push with

Algorithm 2: *FORA* for Whole-Graph SSPPR

Input: Graph G, source node s, probability α, threshold r_{max}
Output: Estimated PPR $\hat{\pi}(s, v)$for all $v \in V$

1 Invoke Algorithm 1 with input parameters G, s, α, and r_{max}; let $r(s, v_i)$, $\pi^{\circ}(s, v_i)$ be the returned residue and reserve of node v_i;
2 Let $r_{sum} = \sum_{v_i \in V} r(s, v_i)$ and $\omega = r_{sum} \cdot \frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2 \cdot \delta}$;
3 Let $\hat{\pi}(s, v_i) = \pi^{\circ}(s, v_i)$ for all $v_i \in V$;
4 **for** $v_i \in V$ *with* $r(s, v_i) > 0$ **do**
5 Let $\omega_i = \lceil r(s, v_i) \cdot \omega / r_{sum} \rceil$;
6 Let $a_i = \frac{r(s, v_i)}{r_{sum}} \cdot \frac{\omega}{\omega_i}$;
7 **for** $i = 1$ *to* ω_i **do**
8 Generate a random walk W from v_i;
9 Let t be the end point of W;
10 $\hat{\pi}(s, t) += \frac{a_i \cdot r_{sum}}{\omega}$;
11 return $\hat{\pi}(s, v_1), \cdots, \hat{\pi}(s, v_n)$;

early termination, and subsequently runs random walks. Similar to BiPPR and HubPPR, FORA utilizes information obtained through Forward Push to significantly cut down the number of required random walks while satisfying the same result quality guarantees. But unlike BiPPR and HubPPR, in FORA there is a single invocation of Forward Push starting from the source node s, while BiPPR and HubPPR invokes numerous backward search operations. The tricky part in FORA is how to combine Forward Push with MC, explained below.

Specifically, the reason that Forward Push with early termination fails to obtain any result quality guarantee is that it uses $\pi^{\circ}(s, v)$ to approximate $\pi(s, v)$, and yet, the two values are not guaranteed to be close. To mitigate this deficiency, we aim to utilize the residue $r(s, v)$ to improve the accuracy of $\pi^{\circ}(s, v)$. Towards this end, we utilize the following result from [2]:

$$\pi(s, t) = \pi^{\circ}(s, t) + \sum_{v \in V} r(s, v) \cdot \pi(v, t), \tag{5}$$

for any s, t, v in G. Our idea is to derive a rough approximation of $\pi(v, t)$ for each node v (denoted as $\pi'(v, t)$), and then combine it with the reserve of each node to compute an estimation of $\pi(s, t)$:

$$\pi(s, t) = \pi^{\circ}(s, t) + \sum_{v \in V} r(s, v) \cdot \pi'(v, t).$$

In particular, we derive $\pi'(v, t)$ by performing a number of random walks from v, and set $\pi'(v, t)$ to the fraction of walks that ends at t.

It remains to answer two key questions in FORA: (i) how many random walks do we need for each node v? and (ii) how should we set the residue threshold r_{max} in Forward Push? It turns out that although the FORA algorithm itself is simple, deriving the proper values for its parameters is rather challenging, since they must optimize efficiency while satisfying the result quality guarantee. In the following, we first present the complete FORA and answer question (i); then we answer question (ii) in Section 3.2.

Algorithm 2 illustrates the pseudo-code of FORA. Given G, a source node s, a probability value α, and a residue threshold r_{max}, FORA first invokes Algorithm 2 on G to obtain a reserve $\pi^{\circ}(s, v_i)$ and a residue $r(s, v_i)$ for each node v_i (Line 1 in Algorithm 2). After that, it computes the total residue of all nodes r_{sum}, based on which it derives a value ω that will be used to decide the number of random walks required from each node v_i (Line 2). Then, it initializes the PPR estimation of each v_i to be $\hat{\pi}(s, v_i) = \pi^{\circ}(s, v_i)$, and it proceeds to inspect the nodes whose residues are larger than zero (Line 3-4).

For each v_i of those nodes, it performs ω_i random walks from v_i, where

$$\omega_i = \left\lceil \frac{r(s, v_i)}{r_{sum}} \cdot \omega \right\rceil.$$

If a random walk ends at a node t, then FORA increases $\hat{\pi}(s, v_i)$ by $\frac{a_i \cdot r_{sum}}{\omega}$, where

$$a_i = \frac{r(s, v_i)}{r_{sum}} \cdot \frac{\omega}{\omega_i}.$$

After all v_i are processed, the algorithm returns $\hat{\pi}(s, v_i)$ as the approximated PPR value for v_i (Line 11).

To explain why FORA can provide accurate results, let us consider the ω_i random walks that it generates from a node v_i. Let $X_j(t)$ be a Bernoulli variable that takes value 1 if the j-th random walk terminates at t, and value 0 otherwise. By definition,

$$\mathbb{E}[X_j] = \pi(v_i, t).$$

Then, based on the definition of ω, ω_i, and a_i, we have

$$\mathbb{E}\left[\frac{r_{sum}}{\omega} \cdot \sum_{j=1}^{\omega_i} \left(a_i \cdot X_j\right)\right] = r(s, v_i) \cdot \pi(v_i, t). \tag{6}$$

Observe that $\frac{r_{sum}}{\omega} \cdot \sum_{j=1}^{\omega_i} \left(a_i \cdot X_j\right)$ is exactly the amount of increment that $\hat{\pi}(s, t)$ receives when FORA processes v_i (see Lines 7-10 in Algorithm 2). We denote this increment as ψ_i. It follows that

$$\mathbb{E}\left[\sum_{i=1}^{n} \psi_i\right] = \sum_{i=1}^{n} r(s, v_i) \cdot \pi(v_i, t). \tag{7}$$

Combining Equations 5 and 7, we can see that FORA returns, for each node v, an estimated PPR $\hat{\pi}(s, v)$ whose expectation equals $\pi(s, v)$. Next, we will show that $\hat{\pi}(s, v)$ is very close to $\pi(s, v)$ with a high probability. For this purpose, we utilize the following concentration bound:

Theorem 3.1 ([7]). *Let $X_1, \cdots, X_\omega$ be independent random variables with*

$$\Pr[X_i = 1] = p_i \quad \textit{and} \quad \Pr[X_i = 0] = 1 - p_i.$$

Let $X = \frac{1}{\omega} \cdot \sum_{i=1}^{\omega} a_i X_i$ with $a_i > 0$, and $\nu = \frac{1}{\omega} \sum_{i=1}^{\omega} a_i^2 \cdot p_i$. Then,

$$\Pr[|X - \mathbb{E}[X]| \geq \lambda] \leq 2 \cdot \exp\left(-\frac{\lambda^2 \cdot \omega}{2\nu + 2a\lambda/3}\right),$$

where $a = \max\{a_1, \cdots, a_\omega\}$. □

To apply Theorem 3.1, let us consider the $\omega' = \sum_{i=1}^{n} \omega_i$ random walks generated by FORA. Let $b_j = a_i$ if the j-th random walk starts from v_i. Then, we have $\max_j b_j = 1$, and $b_j^2 \leq b_j$ for any j. In addition, let $Y_j(t)$ be the a random variable that equals 1 if the j-th walk terminates at t, and 0 otherwise. Then, by Theorem 3.1 and Equations 5 and 7, we have the following result.

Lemma 3.2. *For any node v with $\pi(s,v) > \delta$, Algorithm 2 returns an approximated PPR $\hat{\pi}(s,v)$ that satisfies Equation 1 with at least $1 - p_f$ probability.* □

3.2 Choosing r_{max}

Recall from Sections 2.2 and 3.1 that parameter r_{max} determines how quickly we can terminate Forward Push. A high value for r_{max} leads to low cost for Forward Push (since it can terminate early), but high cost for random walks (since a large number of them are required), and vice versa. Thus, finding the appropriate value of r_{max} requires modelling the overall running time of *FORA*. Recall that, the Forward Push runs in $O\left(\frac{1}{r_{max}}\right)$ time. In addition, the expected time complexity of the random walk phase is $O\left(r_{sum} \cdot \frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2}\right)$, since each random walk takes $O(1)$ expected time to generate. Observe that

$$r_{sum} = \sum_{v_i \in V} r(s,v_i) \leq \sum_{v_i \in V} r_{max} \cdot |N^{out}(v_i)| = m \cdot r_{max}.$$

Therefore, the expected running time of Algorithm 2 is

$$O\left(\frac{1}{r_{max}} + m \cdot r_{max} \cdot \frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2 \cdot \delta}\right).$$

Using the method of Lagrange multipliers, we can see that the above time complexity is minimized when

$$r_{max} = \frac{\epsilon}{\sqrt{m}} \cdot \sqrt{\frac{\delta}{(2\epsilon/3+2)\cdot\log(2/p_f)}}. \tag{8}$$

Accordingly, the expected time complexity of Algorithm 2 becomes

$$O\left(\frac{1}{\epsilon\cdot\sqrt{\delta}}\sqrt{m\cdot(2\epsilon/3+2)\cdot\log(2/p_f)}\right).$$

When $\delta = O(1/n)$, $p_f = O(1/n)$, the above time complexity becomes $O\left(\frac{1}{\epsilon}\sqrt{m\cdot n\cdot\log n}\right)$ for general graphs. When the graph is scale-free, in which case $m/n = O(\log n)$, the time complexity becomes $O\left(\frac{1}{\epsilon}n\cdot\log n\right)$, improving over the MC approach by $1/\epsilon$.

3.3 Indexing Scheme

Based on FORA, we propose a simple and effective index structure to further improve the efficiency of whole-graph SSPPR queries. The basic idea is to pre-compute a number of random walks from each node v, and then store the destination of each walk. During query processing, if FORA requires x performing random walks from v, we would inspect the set S of random walk destinations pre-computed for v, and then retrieve the first x nodes in S. As such, we avoid generating any random walks on-the-fly, which considerably reduces query overheads.

A natural question to ask is: how many random walks should we pre-compute for each node v? To answer this question, we first recall that, when the local update phase of FORA terminates, the residue of each node v is at most $|N^{out}(v)|\cdot r_{max}$. Combining this with Lemma 3.2, we can see that the number of random walks from v required by FORA is

$$\left\lceil |N^{out}(v)|\cdot r_{max}\cdot\frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\cdot\delta}\right\rceil. \tag{9}$$

Since we set r_{max} according to Equation 8, we have

$$\text{Eqn. 9} = \left\lceil |N^{out}(v)|\cdot\frac{1}{\epsilon\cdot\sqrt{m\cdot\delta}}\cdot\sqrt{(2\epsilon/3+2)\cdot\log(2/p_f)}\right\rceil$$

We use $\omega_{max}(v)$ to denote the r.h.s. of the above equation.

In summary, we pre-compute $\omega_{max}(v)$ random walks from each node v, and record the last nodes of those walks in our index structure. The total space overhead incurred is

$$\sum_v \omega_{max}(v) = \sum_v \left\lceil |N^{out}(v)|\cdot\frac{\sqrt{(2\epsilon/3+2)\cdot\log(2/p_f)}}{\epsilon\cdot\sqrt{m\cdot\delta}}\right\rceil \leq n + \frac{\sqrt{m}}{\epsilon\cdot\sqrt{\delta}}\cdot\sqrt{(2\epsilon/3+2)\cdot\log(2/p_f)}.$$

Therefore, we have the following lemma.

Lemma 3.3. *The space consumption of our index structure is*

$$O\left(n + \frac{1}{\epsilon}\sqrt{\frac{m\log(1/p_f)}{\delta}}\right). \tag{10}$$

When $\delta = O(1/n)$, $p_f = O(1/n)$, and $m/n = O(\log n)$, the above space complexity becomes $O\left(\frac{1}{\epsilon}n\cdot\log n\right)$. □

Remark. One may wonder whether we can also pre-compute the Forward Push result for each node, so that we can answer each query by a simple combination of pre-processed Forward Push and random walks, which could lead to higher query efficiency. However, we note that storing the Forward Push results for all nodes incurs significant space overheads. In particular, it requires $O(\min\{n, 1/r_{max}\})$ space for each node, where r_{max} is set according to Equation 8. As such, the total space consumption for preprocessing Forward Push results is

$$O\left(\min\left\{n^2, \frac{n}{\epsilon}\cdot\sqrt{\frac{m\cdot\log(1/p_f)}{\delta}}\right\}\right),$$

which is prohibitive for large graphs. Therefore, we do not store Forward Push results in our index structure.

3.4 Top-k SSPPR

In this section, we discuss how FORA handles approximate top-k SSPPR queries.

Rationale. A straightforward approach to answer a top-k SSPPR query with FORA is to first apply it to perform a whole-graph SSPPR query, and then returns the k nodes with the largest approximate PPR values. However, if we are to satisfy the accuracy requirement described in Definition 2.2, we would need to set the parameters of FORA according to the exact k-th largest PPR value $\pi(s,v_k^*)$, which is unknown in advance. To address this, a naive solution is to conservatively set $\pi(s,v_k^*) = 1/n$, which, however, would lead to unnecessary overheads.

To avoid the aforementioned overheads, we propose a trial-and-error approach as follows. We first assume that $\pi(s,v_k^*)$ is a large value (e.g., 1/2), and we set the parameters of FORA accordingly to perform a whole-graph SSPPR query. After that, we inspect the results obtained to check whether the estimated PPR values are indeed large. If they are not as large as we have assumed, then we re-run FORA with more conservative parameters, and check the new results returned. This process is conducted iteratively, until

Algorithm 3: Top-k FORA

Input: Graph G, source node s, probability α
Output: k nodes with the highest approximate PPR scores

1 **for** $\delta = \frac{1}{2}, \cdots, \frac{1}{n}$ **do**
2 Invoke Algorithm 2 with G, s, α, and r_{max} set by Equation 8 and fail probability $p'_f = \frac{p_f}{n \cdot \log n}$;
3 Let $C = \{v'_1, \cdots, v'_k\}$ be the set that contains the k nodes with the top-k largest lower bounds (from Theorem 3.1);
4 Let $LB(u)$ and $UB(u)$ be the lower and upper bounds of $\pi(s, u)$ (from Theorem 3.1);
5 **if** $UB(v'_i) < (1+\epsilon) \cdot LB(v'_i)$ *for* $i \in [1, k]$ ***and*** $LB(v'_k) \geq \delta$ **then**
6 Let U be the set of nodes $u \in V \setminus C$ such that $UB(u) > (1+\epsilon) \cdot LB(v'_k)$;
7 **if** $\nexists u \in U$ ***such that*** $UB(u) < (1+\epsilon) \cdot LB(u)/(1-\epsilon)$ **then**
8 return $v'_1, v'_2, \cdots, v'_k$ and their estimated PPR;

we are confident that the results from FORA conform to the requirements in Definition 2.2.

Algorithm. Algorithm 3 shows the pseudo-code of the top-k extension of FORA. The algorithm consists of at most $\log n$ iterations. In the i-th iteration, we invoke Algorithm 2 with δ set to $1/2^i$, and the failure probability set to $p'_f = \frac{p_f}{n \log n}$ (Lines 1-2 in Algorithm 3. (The reason for this setting will be explained shortly). After we obtain the results from *FORA*, we compute an upper bound and a lower bound of each node's PPR value, and use them to decide whether the current top-k results are sufficiently accurate (Lines 3-8). If the top-k results are accurate, then we return them as the top-k answers (Line 8); otherwise, we proceed to the next iteration. In the following, we elaborate how the upper and lower bounds of each node's PPR value is derived.

Define $LB_0(v) = 0$ and $UB_0(v) = 1$ for any $v \in V$. We have the following theorem that establishes the lower bound $LB_j(v)$ and upper bound $UB_j(v)$ of $\pi(s, v)$ in the j-th iteration of Algorithm 3:

Theorem 3.4. *In the j-th iteration of Algorithm 3, let ω_j be the ω calculated by FORA (Algorithm 2 Line 2) in this iteration, and $\pi^\circ_j(s, v)$ and $\hat{\pi}_j(s, v)$ be the reserve and estimated PPR of v. Define*

$$\epsilon_j = \sqrt{\frac{3 r_{sum} \cdot \log(2/p'_f)}{\omega_j \cdot \max\{\pi^\circ_j(s, v), LB_{j-1}(s, v)\}}}, \text{ and } \lambda_j = \frac{2/3 \log(2/pf')}{2\omega_j} + \frac{\sqrt{\frac{4}{9} r^2_{sum} \cdot \log^2(2/p'_f) + 8 r_{sum} \cdot \omega_j \cdot \log(2/p'_f) \cdot UB_{j-1}(v)}}{2\omega_j}$$

Then, with at least $1 - p'_f$ probability, the following two inequalities hold simultaneously:

$$\hat{\pi}_j(s, v)/(1+\epsilon_j) \leq \pi(s, v) \leq \hat{\pi}_j(s, v)/(1-\epsilon_j)$$
$$\hat{\pi}_j(s, v) - \lambda_i \leq \pi(s, v) \leq \hat{\pi}_j(s, v) + \lambda_i.$$

Theorem 3.4 enables us to derive tight lower and upper bounds of each node's PPR value in each iteration. In particular, we set

$$UB_j(v) = min\{1, \hat{\pi}_j(s, v)/(1-\epsilon), \hat{\pi}_j(s, v) + \lambda_i\},$$

$$LB_j(v) = \max\{\hat{\pi}_j(s, v)/(1+\epsilon), \hat{\pi}_j(s, v) - \lambda_i, 0\}.$$

With these upper and lower bounds, the following theorem shows that if Lines 5 and 7 in Algorithm 3 holds, then Algorithm 3 returns the answer for the approximate top-k SSPPR query.

Theorem 3.5 (Approximate Top-k). *Let $v'_1, \cdots, v'_k$ be the k nodes with the largest lower bounds in the j-th iteration of Algorithm 3. Let U be the set of nodes $u \in V \setminus C$ such that $UB_j(u) > (1+\epsilon) \cdot LB_j(v'_k)$, If $UB(v'_i) < (1+\epsilon) \cdot LB(v'_i)$ for $i \in [1, k]$, $LB_j(v'_k) \geq \delta$, and there exists no $u \in U$ such that $UB_j(u) < (1+\epsilon) \cdot LB_j(u)/(1-\epsilon)$, then returning $v'_1, \cdots, v'_k$ and their estimated PPR values would satisfy the requirements in Definition 2.2 with at least $1 - j \cdot n \cdot p'_f$ probability.*

Now recall that the number of iterations in Algorithm 3 is $\log n$, and in each iteration, we assume that the upper and lower bounds are correct. Hence, by applying union bound, the failure probability will be at most $n \log n \cdot p'_f$. Note that $p'_f = \frac{p_f}{n \log n}$. The failure probability is hence no more than p_f, and we guarantee that the returned answer has approximation with at least $1 - p_f$ probability.

4 OTHER RELATED WORK

Apart from the methods discussed in Section 2.2 , there exists a plethora of techniques for whole-graph and top-k SSPPR queries. Those techniques, however, are either subsumed by *BiPPR* and *HubPPR* or unable to provide worst-case accuracy guarantees. In particular, a large number of techniques adopt the *matrix-based* approach, which formulates PPR values with the following equation:

$$\boldsymbol{\pi_s} = \alpha \cdot \boldsymbol{e_s} + (1-\alpha) \cdot \boldsymbol{\pi_s} \cdot D^{-1} A, \tag{11}$$

where $\boldsymbol{\pi_s}$ is a vector whose i-th element equals $\pi(s, v_i)$, $A \in \{0, 1\}^{n \times n}$ is the adjacency matrix of G, and $D \in R^{n \times n}$ is a diagonal matrix in which each i-th element on its main diagonal equals the out-degree of v_i. Matrix-based methods typically start from an initial guess of $\boldsymbol{\pi_s}$, and then iteratively apply Equation 11 to refine the initial guess, until converge is achieved. Recent work that adopts this approach [10, 17, 21, 24] propose to decompose the input graph into tree structures or sub-matrices, and utilize the decomposition to speed up the PPR queries. The state-of-the-art approach for the single-source and top-k PPR queries in this line of research work is *BEAR* proposed by Shin et al. [21]. However, as shown in [22], the best of these methods is still inferior to *HubPPR* [22] in terms of query efficiency and accuracy.

There also exist methods that follow similar approaches to the forward search method [2] described in Section 2.2. Berkin et al. [6] propose to pre-compute the Forward Push results from several important nodes, and then use these results to speed up the query performance. Ohsaka et al. [18] and Zhang et al. [23] further design algorithms to update the stored Forward Push results on dynamic graphs. Jeh et al. [13] propose the backward search algorithm, which (i) is the reverse variant of the Forward Push method, and (ii) can calculates the estimated PPRs from all nodes to a target node t. Zhang et al. [23] also design the algorithms to update the stored backward push results on dynamic graphs. Nonetheless, none of these solutions in this category provide approximation guarantees for single-source or top-k PPR queries on directed graphs.

Name	n	m	Type	Linking Site
DBLP	613.6K	2.0M	undirected	www.dblp.com
Web-St	281.9K	2.3M	directed	www.stanford.edu
Pokec	1.6M	30.6M	directed	pokec.azet.sk
LJ	4.8M	69.0M	directed	www.livejournal.com
Orkut	3.1M	117.2M	undirected	www.orkut.com
Twitter	41.7M	1.5B	directed	twitter.com

Table 2: Datasets. ($K = 10^3, M = 10^6, B = 10^9$)

	MC	*BiPPR*	*HubPPR*	*FORA*	*FORA+*
DBLP	12.9	3.5	2.4	0.7	0.07
Web-St	5.0	3.5	1.5	0.03	0.01
Pokec	66.4	23.7	19.2	12.1	0.8
LJ	144.7	57.8	48.3	21.0	1.8
Orkut	241.4	168.7	133.2	46.4	4.9
Twitter	4.6K	3.3K	2.8K	891.5	103.1

Table 3: Whole-graph SSPPR performance (s). ($K = 10^3$)

In addition, there are techniques based on the Monte-Carlo framework. Fogaras et al. [8] propose techniques to pre-store the random walk results, and use them to speed up the query processing. Nonetheless, the large space consumption of the technique renders it applicable only on small graphs. Bahmani et al. [4] and Sarma et al. [20] investigate the acceleration of the Monte-Carlo approach in distributed environments. Lofgren et al. propose *FastPPR* [16], which significantly outperforms the Monte-Carlo method in terms of query time. However, *FastPPR* in turn is subsumed by *BiPPR* [15] in terms of query efficiency. In [14], Lofgren further proposes to combine a modified version of Forward Push, random walks, and the back search algorithm to reduce the processing time of pairwise PPR queries. Nevertheless, the time complexity of the method remains unclear, since [14] does not provide any theoretical analysis on the asymptotic performance of the method.

Finally, Ref. [3, 5, 9–11, 15] present studies on the top-k PPR queries. Gupta et al. [11] propose to use Forward Push to return the top-k answers. However, their solutions do not provide any approximation guarantee. Avrachenkov et al. [3] study how to use Monte-Carlo approach to find the top-k nodes. Nevertheless, the solution does not return estimated PPR values and does not provide any worst-case assurance. Fujiwara et al. [9, 10] and Shin et al.[21] investigate how to speed up the top-k PPR queries with the matrix decomposition approach. These approaches either cannot scale to large graphs or do not provide approximation guarantees.

5 EXPERIMENTS

In this section, we experimentally evaluate *FORA* and its index-based variant, referred to as *FORA+*, against the states of the art. All experiments are conducted on a Linux machine with an Intel 2.6GHz CPU and 64GB memory.

5.1 Experimental Settings

Datasets and query sets. We use 6 real graphs: *DBLP*, *Web-St*, *Pokec*, *LJ*, *Orkut*, and *Twitter*, which are benchmark datasets used in recent work [15, 22]. Table 2 summarizes the statistics of the data. For each dataset, we choose 50 source nodes uniformly at random, and we generate an SSPPR query from each chosen node. In addition, we also generate 5 top-k queries from each source node, with k varying in $\{100, 200, 300, 400, 500\}$. Note that the maximum k is set to 500 in accordance to Twitter's Who-To-Follow service [12], whose first step requires deriving top-500 PPR results.

Methods. For whole-graph SSPPR queries, we compare our proposed *FORA* and *FORA+* against three methods: (i) the *Monte-Carlo* approach, dubbed as *MC*; (ii) the optimized *BiPPR* for SSPPR queries described in Appendix B.1; (iii) *HubPPR*, which is the indexed version of *BiPPR*. For fair comparison, the index size of *HubPPR* is set to be the same as that of *FORA+*. For top-k SSPPR queries, we compare our algorithm with the existing approximate solutions: the single-source MC, the single-source *BiPPR*, as well as the top-k algorithm for *HubPPR* in [22]. We also include the *Forward Push* as a baseline for top-k SSPPR queries, and we tune its parameter r_{max} on each dataset separately, so that its precision for top-k PPR queries is the same as *FORA* on each dataset. Following previous work[15, 16, 22], we set $\delta = 1/n, p_f = 1/n$, and $\epsilon = 0.5$.

5.2 Whole-Graph SSPPR Queries

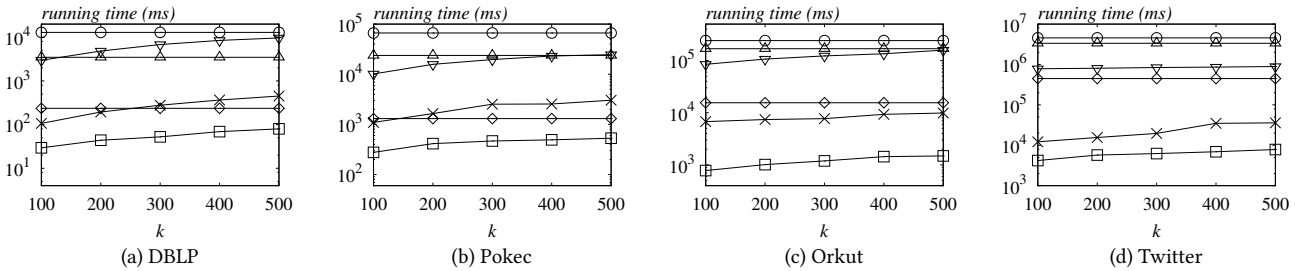
In our first set of experiments, we evaluate the efficiency of each method for whole-graph SSPPR queries. Table 3 reports the average query time of each method. Observe that both *FORA* and *BiPPR* achieve better query performance than *MC*, which is consistent with our analysis that the time complexity of *FORA* and *BiPPR* is better than that of *MC*. Moreover, *FORA* is at least 3 times faster than *BiPPR* on most of the datasets.

The reason, as we explain in Section 3.2, is that *BiPPR* either degrades to the *MC* approach when the backward threshold is large, or requires conducting a backward search from each node v in G, even if $\pi(s, v)$ is extremely small. In contrast, *FORA* avoids degrading to *MC* and tends to omit nodes with small PPR values, which helps improve efficiency.

In addition, *FORA+* achieves significant speedup over *FORA*, and is around 10 times faster than the latter on most of the datasets. The *HubPPR* also improves over *BiPPR*, but the improvement is far less than what *FORA+* achieves over *FORA*. Moreover, even without any index, *FORA* is still more efficient than *HubPPR*.

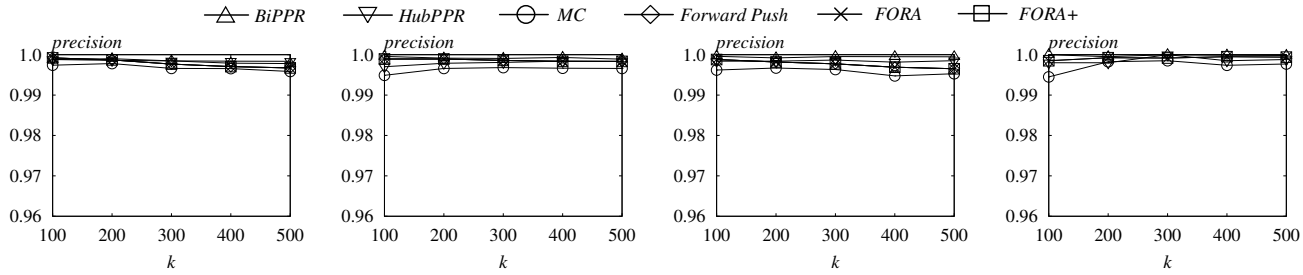
5.3 Top-k SSPPR Queries

In our second set of experiments, we evaluate the efficiency and accuracy of each method for top-k SSPPR queries. Figures 1 reports the average query time of each method on four representative datasets: *DBLP*, *Pokec*, *Orkut*, and *Twitter*. (The results on the other two datasets are qualitatively similar, and are omitted due to the space constraint.) Note that the y-axis is in log-scale. The main observation is that *FORA* and *FORA+* both considerably outperform competitors. In particular, *FORA* is up to two orders of magnitude faster than *MC* and *BiPPR*. This is expected since our top-k algorithm applies an iterative approach to refine the top-k answers and terminates immediately whenever the answer could provide the desired approximation guarantee. *HubPPR*'s top-k algorithm has a similar early-termination mechanism, but it is still outperformed by our *FORA* by more than an order of magnitude, since



(a) DBLP (b) Pokec (c) Orkut (d) Twitter

Figure 1: Top-k SSPPR query efficiency: varying k.



(a) DBLP (b) Pokec (c) Orkut (d) Twitter

Figure 2: Top-k SSPPR query accuracy: varying k.

Datasets	Preprocessing time (sec)		Space overhead of *HubPPR* & *FORA+*	
	HubPPR	*FORA+*	Index size	Graph size
DBLP	28.7	4.9	90.8MB	18.4MB
Web-St	11.8	2.0	45.5MB	10.4MB
Pokec	112.3	38.6	411.9MB	130.8MB
LJ	307.9	97.1	1.1GB	295.2MB
Orkut	582.2	204.9	1.6GB	950.1MB
Twitter	5849.0	3335.1	12.6GB	6.2GB

Table 4: Preprocessing costs.

HubPPR inherits the deficiences of *BiPPR* on SSPPR queries. Recall that *Forward Push* provides no approximation guarantee, and we tune the r_{max} on each dataset separately, so that it provides the same precison for top-500 SSPPR queries as our *FORA* algorithm does. Observe that, when both algorithms provide the identical precision, the performance of the proposed *FORA* and *Forward Push* is roughly comparable on small and medium size graphs, e.g., on *DBLP*, *Pokec*, and *Orkut*. When the graph size increases, however, *FORA* dominates the *Forward Push* in terms of the query efficiency. In particular, on *Twitter*, *FORA* is faster than *Forward Push* by an order of magnitude.

To compare the accuracy of the top-k results returned by each method, we first calculate the ground-truth answer of the top-k queries using the *Power Iteration* [19] method with 100 iterations. Afterwards, we evaluate the top-k results of each algorithm by their precision with respect to the ground truth. Note that the precision and recall are the same for the top-k SSPPR queries. Figure 2 shows the accuracy of the top-k query algorithms on four datasets: *DBLP*, *Pokec*, *Orkut*, and *Twitter*. Observe that all methods provide high precisions, and *FORA*, *FORA+*, *BiPPR*, and *HubPPR* are all slightly more accurate than *MC*.

5.4 Preprocessing Costs

Finally, we inspect the preprocessing costs of the two index-based methods, *FORA+* and *HubPPR*. Recall that we set the index size of *HubPPR* the same as the proposed *FORA+*. As shown in Table 4, the preprocessing time of *FORA* and *HubPPR* are both moderate, and the cost of *FORA* is much smaller than that of *HubPPR*. In addition, even on the largest dataset Twitter, *FORA+* can finish the index construction in less than an hour. Table 4 also reports the index size of *FORA+* and *HubPPR*. We also add the space consumption of the input graph as the reference values. As we can observe, the index size of *FORA+* is no more than 4 times the original graph in general. In particular, on *Orkut*, the index size of *FORA+* is only 1.7 times the input graph, and yet, as we show in Sections 5.2 and 5.3, it can speed up the query efficiency of whole-graph and top-k SSPPR queries by around 10 times. This demonstrates the effectiveness and efficiency of our indexing scheme.

6 CONCLUSION

We present FORA, a novel algorithm for approximate single-source personalized PageRank computation. The main ideas include (i) combining Monte-Carlo random walks with Forward Push in a non-trivial and optimized way (ii) pre-computing and indexing random walk results and (iii) additional pruning based on top-k selection. Compared to existing solutions, FORA involves a reduced number of random walks, avoids expensive backward searches, and provides rigorous guarantees on result quality. Extensive experiments demonstrate that FORA outperforms existing solutions by a large margin, and enables fast responses for top-k SSPPR searches on very large graphs with little computational resource.

7 ACKNOWLEDGMENTS

This research is supported by the DSAIR center at NTU, Singapore, as well as grants MOE2015-T2-2-069 from MOE, Singapore,

NSFC.61502503 from NSCF, China, and NPRP9-466-1-103 from QNRF, Qatar.

REFERENCES

[1] Reid Andersen, Christian Borgs, Jennifer T. Chayes, John E. Hopcroft, Vahab S. Mirrokni, and Shang-Hua Teng. 2007. Local Computation of PageRank Contributions. In *WAW*. 150–165.

[2] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. 475–486.

[3] Konstantin Avrachenkov, Nelly Litvak, Danil Nemirovsky, Elena Smirnova, and Marina Sokol. 2011. Quick Detection of Top-k Personalized PageRank Lists. In *WAW*. 50–61.

[4] Lars Backstrom and Jure Leskovec. 2011. Supervised random walks: predicting and recommending links in social networks. In *WSDM*. 635–644.

[5] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. 2011. Fast personalized PageRank on MapReduce. In *SIGMOD*. 973–984.

[6] Pavel Berkhin. 2005. Survey: A Survey on PageRank Computing. *Internet Mathematics* 2, 1 (2005), 73–120.

[7] Fan R. K. Chung and Lincoln Lu. 2006. Survey: Concentration Inequalities and Martingale Inequalities: A Survey. *Internet Mathematics* 3, 1 (2006), 79–127.

[8] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358.

[9] Yasuhiro Fujiwara, Makoto Nakatsuji, Hiroaki Shiokawa, Takeshi Mishima, and Makoto Onizuka. 2013. Efficient ad-hoc search for personalized PageRank. In *SIGMOD*. 445–456.

[10] Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka. 2012. Efficient personalized pagerank with accuracy assurance. In *KDD*. 15–23.

[11] Manish S. Gupta, Amit Pathak, and Soumen Chakrabarti. 2008. Fast algorithms for topk personalized pagerank queries. In *WWW*. 1225–1226.

[12] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. Wtf: The who to follow service at twitter. In *WWW*. 505–514.

[13] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *WWW*. 271–279.

[14] Peter Lofgren. 2015. Efficient Algorithms for Personalized PageRank. *CoRR* abs/1512.04633 (2015).

[15] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. 2016. Personalized pagerank estimation and search: A bidirectional approach. In *WSDM*. 163–172.

[16] Peter A Lofgren, Siddhartha Banerjee, Ashish Goel, and C Seshadhri. 2014. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *KDD*. 1436–1445.

[17] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. 2014. Computing personalized PageRank quickly by exploiting graph structures. *PVLDB* 7, 12 (2014), 1023–1034.

[18] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. 2015. Efficient PageRank Tracking in Evolving Networks. In *SIGKDD 2015*. 875–884.

[19] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).

[20] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. 2013. Fast Distributed PageRank Computation. In *ICDCN*. 11–26.

[21] Kijung Shin, Jinhong Jung, Lee Sael, and U. Kang. 2015. BEAR: Block Elimination Approach for Random Walk with Restart on Large Graphs. In *SIGMOD*. 1571–1585.

[22] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. 2016. HubPPR: Effective Indexing for Approximate Personalized PageRank. *PVLDB* 10, 3 (2016), 205–216. http://www.vldb.org/pvldb/vol10/p205-wang.pdf

[23] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate Personalized PageRank on Dynamic Graphs. In *KDD*. 1315–1324.

[24] Fanwei Zhu, Yuan Fang, Kevin Chen-Chuan Chang, and Jing Ying. 2013. Incremental and Accuracy-Aware Personalized PageRank through Scheduled Approximation. *PVLDB* 6, 6 (2013), 481–492.

APPENDIX

A PROOF OF THEOREM

Proof of Lemma 3.2. Firstly, let X_j be the j-th random walk. Define $Y' = \frac{1}{\omega'}\sum_{j=1}^{\omega'} b_j X_j$, and $\nu = \frac{1}{\omega'}\sum_{j=1}^{\omega'} b_j^2 \mathbb{E}[X_j]$. Let $a = \max\{b_1, \cdots, b_{\omega'}\}$. By definition, $b_j^2 \le 1$, and hence, $\nu \le \mathbb{E}[Y']$ and $a \le 1$. By Theorem 3.1, $\Pr[|Y' - \mathbb{E}[Y']| \ge \lambda] \le 2 \cdot \exp\left(-\frac{\lambda^2 \cdot \omega'}{2\nu + 2a\lambda/3}\right)$. Apply $\nu \le \mathbb{E}[Y']$, we have

$$\Pr[|Y' - \mathbb{E}[Y']| \ge \lambda] \le 2 \cdot \exp\left(-\frac{\lambda^2 \cdot \omega'}{2\mathbb{E}[Y'] + 2a\lambda/3}\right).$$

By Equation 6 and $\mathbb{E}[Y'] \le \frac{\omega}{\omega' \cdot r_{sum}} \cdot \pi(s,t)$,

$$\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \frac{\omega' \cdot r_{sum}}{\omega}\lambda] \le 2 \cdot \exp\left(-\frac{\lambda^2 \cdot \omega'}{2\frac{\omega \cdot \pi(s,t)}{\omega' \cdot r_{sum}} + 2a\lambda/3}\right).$$

Let $\lambda = \frac{\omega \cdot \epsilon \cdot \pi(s,t)}{\omega' \cdot r_{sum}}$, we have

$$\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \epsilon \cdot \pi(s,t)] \le 2 \cdot \exp\left(-\frac{\epsilon^2 \cdot \omega \cdot \pi(s,t)}{r_{sum} \cdot (2 + 2a \cdot \epsilon/3)}\right).$$

As we only consider approximation for $\pi(s,t) > \delta$ and $a \le 1$,

$$\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \epsilon \cdot \pi(s,t)] \le 2 \cdot \exp\left(-\frac{\epsilon^2 \cdot \omega \cdot \delta}{r_{sum} \cdot (2 + 2\epsilon/3)}\right).$$

Since $\omega = r_{sum} \cdot \frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2 \cdot \delta}$, we can conclude that

$$\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \epsilon \cdot \pi(s,t)] \le p_f$$

holds for $\pi(s,t) > \delta$. Also notice that the target t is arbitrarily chosen, and we can derive this bound for all nodes $v \in V$. Hence, the returned answer for the single-source PPR query satisfies Definition 2.1, which finishes the proof. □

Proof of Theorem 3.4. Given ω_j, we can derive that:

$$\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \epsilon \cdot \pi(s,t)] \le 2 \cdot \exp\left(-\frac{\epsilon^2 \cdot \omega_j \cdot \pi(s,t)}{2 + 2a \cdot \epsilon/3}\right).$$

Since $a \le 1$ and $\pi(s,t) \ge \pi^\circ(s,t)$, and $\pi(s,t) \ge LB_{j-1}(t)$, we can derive that:

$$\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \epsilon \cdot \pi(s,t)] \le 2\exp\left(-\frac{\epsilon^2 \cdot \omega_j \cdot \max\{\pi_j^\circ(s,v), LB_{j-1}(s,v)\}}{2 + 2\epsilon/3}\right).$$

Let p_f' equal the RHS of the above inequality. We have $\epsilon \ge \sqrt{\frac{3\log(2/p_f')}{\omega_j \cdot \max\{\pi_j^\circ(s,v), LB_{j-1}(s,v)\}}}$.

By setting $\epsilon_j = \sqrt{\frac{3\log(2/p_f')}{\omega_j \cdot \max\{\pi_j^\circ(s,v), LB_{j-1}(s,v)\}}}$, we can derive that $\hat{\pi}_j(s,v)/(1+\epsilon_j) \le \pi(s,v) \le \hat{\pi}_j(s,v)/(1-\epsilon_j)$ holds with $1 - p_f'$ probability. Similarly, we have

$$\Pr[|\pi(s,t) - \hat{\pi}(s,t)| \ge \epsilon \cdot \pi(s,t)] \le 2\exp\left(-\frac{\lambda^2 \cdot \omega}{r_{sum} \cdot (2\pi(s,t) + 2\lambda/3)}\right)$$

Let p_f' equal the RHS of the above inequality, and note that $\pi(s,t) \le UB_{j-1}(t)$. This helps us derive the designed bound for λ_j. □

Proof of Theorem 3.5. We apply a similar technique in [22]. If $LB_j(v_i') \cdot (1+\epsilon) > UB_j(v_i')$. Then, it can be derived that

$$\hat{\pi}(v_i') \le UB_j(v_i') \le LB_j(v_i') \cdot (1+\epsilon) \le (1+\epsilon) \cdot \pi(s, v_i').$$
$$\hat{\pi}(v_i') \ge LB_j(v_i') \ge UB_j(v_i')/(1+\epsilon) \ge (1-\epsilon) \cdot \pi(s, v_i').$$

Hence, $v_1', \cdots, v_k'$ satisfy Equation 2. Let $v_1, \cdots, v_k$ be the k nodes that have the top-k exact PPR values. Assume that all bounds are correct, then as $LB(v_l') \ge \delta$, it indicates that the top-k PPR values are no smaller than δ. In this case, all the top-k nodes should

satisfy ϵ-approximation guarantee, i.e., they satisfy that $UB(v_i) < (1+\epsilon) \cdot LB(v_i)/(1-\epsilon)$.

Let $UB'_j(1), UB'_j(2), \cdots UB'_j(k)$ be the top-k largest PPR upper bounds in the j-th iteration. Note that, the i-th largest PPR satisfy that $UB'_j(i) \geq \pi(s, v_i) \geq LB_j(s, v'_i)$.

Now assume that one of the upper bounds say $UB'_j(i)$ is not from $UB(v_1), \cdots, UB(v_k)$. If it satisfies that $LB_j(v'_k) \cdot (1+\epsilon) \geq UB'_j(i)$, it indicates that $LB_j(v'_i) \cdot (1+\epsilon) \geq UB'_j(i)$ for all nodes. Hence, we update the node whose upper bound is minimum among $UB_j(v'_1), \cdots, UB_j(v'_1)$, we can still guarantee that $LB_j(v'_i) \cdot (1+\epsilon) > UB_j(v'_i)$. We repeat this process until $UB_j(v'_1), \cdots, UB_j(v'_k)$ are the top-k upper bounds. On the other hand, let U be the set of nodes such that, the node $u \in U$ satisfies that $UB_j(u) < LB_j(v'_k)$. Then it is still possible that these nodes are from the exact top−k answers. However, recall that if a node $u \in U$ is from the top-k, it should satisfy that $UB(u) < (1+\epsilon) \cdot LB(u)/(1-\epsilon)$. As a result, if there exists no node $u \in U$ such that $UB(u) < (1+\epsilon) \cdot LB(u)/(1-\epsilon)$, then no node $u \in U$ is from the top-k answers, in which case it will not affect the approximation guarantee.

Afterwards, we proceed a bubble sort on the top-k upper bounds in decreasing order. If we replace two upper bounds $UB_j(v'_x)$ and $UB_j(v'_y)$ with $x < y$, then $UB_j(v'_x) < UB_j(v'_y)$. Also $LB_j(v'_x) > LB_j(v'_y)$ from the definition. As

$$UB_j(v'_x)/LB_j(v'_y) < UB_j(v'_y)/LB_j(v'_y) \leq (1+\epsilon)$$
$$UB_j(v'_y)/LB_j(v'_x) < UB_j(v'_y)/LB_j(v'_y) \leq (1+\epsilon)$$

When the sort finishes, the inequations still hold. We then have

$$UB'_j(1) \leq (1+\epsilon) \cdot LB_j(v'_i) \cdots, UB'_j(k) \leq (1+\epsilon) \cdot LB_j(v'_k).$$

Also note that

$$\hat{\pi}(v'_i) \geq LB_j(v'_i) \geq \frac{1}{1+\epsilon} UB'_j(i) \geq (1-\epsilon) \cdot \pi(s, v_i),$$

for all $i \in [1, k]$. So, the answer provides approximation guarantee if the bounds from the first to the j-th iteration are all correct. By applying the union bound, we can obtain that the approximation is guaranteed with probability at least $1 - n \cdot j \cdot p'_f$.

B EXTENSIONS

B.1 Extending *BiPPR* to Whole-Graph SSPPR

Recall from Section 2.2 that in BiPPR, it includes both a forward phase and a backward phase. It is proved in [1] that the amortized time complexity for the backward phase is $O\left(\frac{m}{n \cdot r_{max}}\right)$, and in [15], it shows that the forward phase requires $O\left(\frac{r_{max} \cdot \log(1/p_f)}{\epsilon^2 \cdot \delta}\right)$ time, given the backward phase threshold r_{max}. Afterwards, they choose $r_{max} = O\left(\epsilon \cdot \sqrt{\frac{m \cdot \delta}{n \cdot \log(1/p_f)}}\right)$ to minimize the time complexity for the pairwise PPR query, which is $O\left(\frac{1}{\epsilon}\sqrt{\frac{m \cdot \log(1/p_f)}{n \cdot \delta}}\right)$. To apply BiPPR for whole-graph SSPPR queries, a straightforward approach is to use it to answer n point-to-point PPR queries (i.e., from s to every other node). This, however, leads to a total time complexity of $O\left(\frac{1}{\epsilon}\sqrt{\frac{mn \cdot \log(1/p_f)}{\delta}}\right)$, which is a factor of $\sqrt{n}$ larger than that of the whole-graph SSPPR FORA.

To improve this, we observe that the n point-to-point PPR queries share the same forward phase, and hence, we can conduct the forward phase once and then re-use its results for all n backward phase. In addition, to reduce the total cost of n backward phases, we can set r_{max} to a larger value; although it would require more random walks to be generated in the forward phase, the tradeoff is still favorable as the overhead of the forward phase has been significantly reduced by the re-usage of results. Since the backward phase (for all target nodes) has a cost of $O\left(\frac{m}{r_{max}}\right)$, it can be verified that, by setting $r_{max} = O\left(\epsilon \cdot \sqrt{\frac{m \cdot \delta}{\log(1/p_f)}}\right)$, the expected time complexity of this optimized version of BiPPR (for SSPPR queries) is

$$O\left(\frac{1}{\epsilon \cdot \sqrt{\delta}} \sqrt{m \cdot \log(1/p_f)}\right),$$

which is identical to that of single-source FORA.

However, as we show in Section 5, the optimized *BiPPR* is significantly outperformed by Whole-Graph SSPPR FORA. The reason is that, even after the aforementioned optimization, BiPPR either degrades to MC when r_{max} is large, or still requires performing a backward phase from each node v in G, even if $\pi(s, v)$ is extremely small and can be omitted. In contrast, single-source FORA does not suffer from these deficiencies, and avoid examining nodes with very small PPR values. Instead, it performs a forward search phase, followed by a number of random walks from the nodes visited in the search; this process tends to avoid examining nodes with very small PPR values, since those nodes are unlikely to be visited by the forward push or the random walks.

B.2 Extending FORA to Source Distributions

In many real applications of the personalized PageRank, the source s can be a distribution (e.g., on a set of bookmark pages) instead of a single node. We show that our algorithms for single-source-node *FORA* can be extended to the case of arbitrary source distributions.

Let σ be the node distribution that the source node s is sampled from. For any target node t, its personalized PageRank with respect to σ is defined as [15]:

$$\pi(\sigma, t) = \sum_{v \in V} \sigma(v) \cdot \pi(v, t),$$

where $\sigma(v)$ is the probability that a sample from σ equals v. To apply our algorithms, we modify Line 1 of Algorithm 1 to set the initial residue of each node v as $\sigma(v)$. Let $\pi^\circ(\sigma, v)$ (resp. $r(\sigma, v)$) denote the reserve (resp. residue) of node v in the modified version of Forward Push. Then, it is easy to prove that the following invariant holds for the modified version of Forward Push:

$$\pi(\sigma, t) = \pi^\circ(\sigma, t) + \sum_{v \in V} r(\sigma, v) \cdot \pi(v, t).$$

In particular, the initial states satisfy the above invariant, and by induction, it can be proved that the invariant still holds after every push operation. Given the above invariant, our algorithms can be applied to compute $\pi(\sigma, t)$ without compromising their asymptotic guarantees. Besides, the indexing scheme presented in Section 3.3 is still applicable, since the maximum number of random walks required for each node is identical to that in the single-source-node algorithms.