Node-Centric Random Walk for Fast Index-Free Personalized PageRank

Kohei Tsuchida
Graduate School of Science and Technology
Keio University
nora@inl.ics.keio.ac.jp

Naoki Matsumoto
Research Institute for Digital Media Content
Keio University
naokimatsumoto@dmc.keio.ac.jp

Kunitake Kaneko
Faculty of Science and Technology
Research Institute for Digital Media Content
Keio University
kaneko@inl.ics.keio.ac.jp

***Abstract*—Personalized PageRank (PPR) is a popular graph computation in various real-world applications. Since massive real-world graphs are evolving rapidly, PPR computation methods require index-free and fast. In general, index-free methods go through Forward Push phase and random walk Monte-Carlo simulation phase respectively. While existing methods have succeeded in accelerating the Forward Push phase, there is a space for running-time improvements in the second phase that performs a large number of sequential random walks. Through this sequential process, each random walk needs to obtain neighbor nodes for every single step, which causes redundant operation at each node as a result. Our proposal is a node-centric random walk that aggregates random walks at each node and minimizes the total number of obtaining neighbor nodes in the second phase. Most of the random walks can be aggregated keeping theoretical guarantees because they do not need to memorize the starting node. In addition, we review the expected running time of random walk Monte-Carlo simulation focusing on the total number of obtaining neighbor nodes. We conducted extensive experiments using four real-world graphs. Experimental results showed that our proposed method is up to 3.3x faster than the existing methods.**

***Index Terms*—Personalized PageRank, Random Walk, Index-Free**

I. Introduction

Personalized PageRank (PPR) [1] is one of the most popular graph computations to measure the proximity of nodes. Of our particular interest is the single-source PPR (SSPPR) among several PPR variants, such as single-target PPR, pairwise PPR, and fully PPR. Given a graph $G = (V, E)$, a termination probability α, a source node s, and a target node t, the SSPPR score $\pi(s, t)$ is defined as the probability that an α-decay random walk starting from s terminates at t. When performing an α-decay random walk, a random walker starting from s terminates at the current node with probability α or moves to a neighbor node with probability $1 - \alpha$. By this definition, the SSPPR scores can also be regarded as the relative importance of all nodes in G with respect to s.

SSPPR has various real-world applications, such as spam detection [2], link prediction [3], social recommendation [4], community detection [5]–[7], graph learning [8]–[10], and so on. Nowadays, real-world massive graphs are evolving rapidly. In this scenario, index-oriented SSPPR computation methods [11]–[13] are impractical because they have to hold huge indices and update them frequently. Therefore, SSPPR computation methods require index-free and fast.

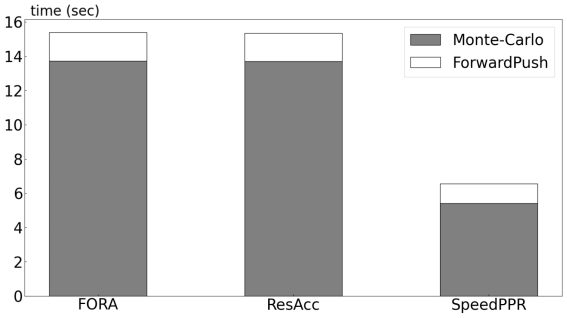


Fig. 1: *MC* is the bottleneck of existing index-free methods: FORA [14], ResAcc [15], and SpeedPPR [16]. We measured their running time using the Pokec social network dataset [17].

Unfortunately, existing index-free methods are still suffering from computational inefficiency. The state-of-the-art algorithm goes through Forward Push (*FP*) phase and random walk Monte-Carlo simulation (*MC*) phase respectively. *FP* is a fast solution without any accuracy guarantees, and on the other hand, *MC* is a slow solution providing rigorous accuracy guarantees. Although existing methods [14]–[16] have succeeded in further accelerating the *FP* phase, they still need too much time to perform a large number of random walks in the *MC* phase. Existing methods perform random walks sequentially because they have to memorize the starting node of each random walk for theoretical guarantees. Through this sequential process, each random walk needs to obtain neighbor nodes for every single step to decide the next destination. Therefore, the total number of obtaining neighbor nodes is excessively large, which is a major cause of computational inefficiency. As a result, *MC* is the bottleneck of existing methods as shown in Figure 1.

Motivated by this, we propose a node-centric random walk. The proposed method aggregates random walks at each node and minimizes the total number of obtaining neighbor nodes in the *MC* phase. Note that most of the random walks can be aggregated keeping theoretical guarantees because they do not need to memorize the starting node. In addition, we review

Notation	Description
$G = (V, E)$	The input graph G with node set V and edge set E
n	The number of nodes in G
m	The number of edges in G
$N_{out}(v)$	The out-neighbor nodes of a node v
$N_{in}(v)$	The in-neighbor nodes of a node v
$d_{out}(v)$	The out-degree of a node v
$d_{in}(v)$	The in-degree of a node v
$\pi(s, v)$	The SSPPR score of a node v
$r(s, v)$	The residue of a node v
$\hat{\pi}(s, v)$	The reserve of a node v
r_{max}	The residue threshold
r_{sum}	The sum of all nodes' residues
α	The random walk termination probability
δ, ϵ, p_f	The parameters of SSPPR queries

TABLE I: Frequently used notations

the expected running time of our node-centric random walk. By focusing on the total number of obtaining neighbor nodes, we establish a new expected running time. Our contributions are summarized as follows.

- We propose a node-centric random walk that aggregates random walks at each node and minimizes the total number of obtaining neighbor nodes in the *MC* phase.
- We review the expected running time of the *MC* phase and establish a new expected running time.
- We conducted extensive experiments using four real-world graphs. Experimental results showed that our proposed method is up to 3.3x faster than the existing methods.

The rest of this paper is organized as follows. Section II states the problem definition. Section III discusses related work. Section IV describes the details of our proposed method. We evaluate the proposed method through extensive experiments using real-world graphs in Section V. Finally, we conclude this paper in Section VI.

II. Preliminaries

A. Problem Definition

Let $G = (V, E)$ be a directed unweighted graph with a set of nodes V and a set of edges E. $n = |V|$, $m = |E|$ are the number of nodes and edges, respectively. For an undirected graph, we convert every undirected edge (u, v) into two directed edges (u, v) and (v, u). Let $N_{in}(v)$ (resp. $N_{out}(v)$) denote a set of in-neighbor nodes (resp. out-neighbor nodes) of $v \in V$, and let $d_{in}(v)$ (resp. $d_{out}(v)$) denote in-degree (resp. out-degree) of $v \in V$. Given a source node $s \in V$ and a termination probability α, an α-decay random walk starting from s terminates at the current node with α probability or moves to an out-neighbor node with $1 - \alpha$ probability. The SSPPR score $\pi(s, t)$ of t with respect to s is defined as the probability that an α-decay random walk starting from s terminates at t. In this paper, we focus on the Approximate SSPPR query, and hereafter, we refer to Approximate SSPPR as SSPPR. Table I summarizes the notations we frequently use in this paper.

Algorithm 1: Forward Push

Input: Graph G, source node s, termination probability α, threshold r_{max}
Output: residue $r(s, t)$ and reserve $\hat{\pi}(s, t)$ for all $t \in V$

1 $\hat{\pi}(s, t) \leftarrow 0$ and $r(s, t) \leftarrow 0$ for all $t \in V$;
2 $r(s, s) \leftarrow 1$;
3 **while** $\exists t \in V$ *such that* $r(s, t) > d_{out}(t) \cdot r_{max}$ **do**
4 $\quad \hat{\pi}(s, t) \leftarrow \hat{\pi}(s, t) + \alpha \cdot r(s, t)$;
5 $\quad$ **for** *each* $u \in N_{out}(t)$ **do**
6 $\quad\quad r(s, u) \leftarrow r(s, u) + (1 - \alpha) \cdot \frac{r(s,t)}{d_{out}(t)}$;
7 $\quad r(s, t) \leftarrow 0$;
8 **return** $r(s, t)$ and $\hat{\pi}(s, t)$ for all $t \in V$;

Definition 1 (Approximate SSPPR)**.** Given a graph $G = (V, E)$, a source node s, a threshold δ, an error bound ϵ, and a failure probability p_f, an Approximate SSPPR query returns the estimated SSPPR score $\hat{\pi}(s, t)$ for all $t \in V$, such that for any $\pi(s, t) > \delta$,

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

holds with at least $1 - p_f$ probability.

B. Random walk Monte-Carlo simulation

Random walk Monte-Carlo simulation (*MC*) is a classic and straightforward solution to answer SSPPR queries [18], [19]. *MC* performs ω random walks from a given source node s, and records the fraction that the number of random walks terminate at t. Then, *MC* uses its fraction to estimate $\hat{\pi}(s, t)$. To satisfy Definition 1, *MC* needs to perform $\omega = \Omega\left(\frac{(2 \cdot \epsilon/3+2) \cdot \log(2/p_f)}{\epsilon^2 \cdot \delta}\right)$ random walks [18]. Assume that a given graph is scale-free, where $m = O(n \cdot \log n)$, the expected running time is bounded by $O(\frac{\log(1/p_f)}{\epsilon^2 \cdot \delta})$, which is infeasible with massive real-world graphs.

C. Forward Push

Forward Push (*FP*) [20] is a local update method. *FP* simulates random walks in a deterministic way by repeatedly pushing the probability mass to neighbor nodes. Algorithm 1 shows the pseudo-code of *FP*. Given a graph G, a source node s, a termination probability α, and a threshold r_{max}, *FP* maintains residue $r(s, t)$ and reserve $\hat{\pi}(s, t)$ for each node $t \in V$. At beginning, *FP* initializes residue $r(s, t)$ and reserve $\hat{\pi}(s, t)$ (Lines 1-2). A node t becomes active when $r(s, t) > r_{max} \cdot d_{out}(t)$. An active node t updates its own $\hat{\pi}(s, t)$ by converting α portion of $r(s, t)$, and transfers $1 - \alpha$ portion of $r(s, t)$ to $r(s, u)$ where $u \in N_{out}(t)$ (Lines 3-7). When there are no active nodes, $\hat{\pi}(s, t)$ is returned as the t's SSPPR score (Line 8). The expected running time of *FP* is $O(\frac{1}{\alpha \cdot r_{max}})$. However, *FP* cannot provide any theoretical guarantees.

III. Related work

FORA [14]. FORA is the first method to combine *FP* and *MC*. FORA first invokes *FP* with early termination and subsequently performs random walks to obtain theoretical guarantees. FORA utilizes the following invariant [20]:

$$\pi(s,t) = \pi(s,t)^{\circ} + \sum_{v \in V} r(s,v) \cdot \pi(v,t), \tag{2}$$

where $\pi(s,t)^{\circ}$ is the reserve of a node t after the *FP* phase. Since computing $\pi(v,t)$ for all nodes $v \in V$ is infeasible, FORA computes $\hat{\pi}(s,t)$ as follows:

$$\hat{\pi}(s,t) = \pi(s,t)^{\circ} + \sum_{v \in V} r(s,v) \cdot \pi(v,t)', \tag{3}$$

where $\pi(v,t)'$ can be obtained by *MC*. In the *MC* phase, each node v performs $\lceil r(s,v) \cdot \omega \rceil$ random walks, where $\omega = \Omega\left(\frac{(2\cdot\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\cdot\delta}\right)$. Recall that the expected running time of *FP* is $O(\frac{1}{\alpha \cdot r_{max}})$ and *MC* in FORA has at most $O\left(\frac{m \cdot r_{max} \cdot \omega}{\alpha}\right)$ running time. By setting $r_{max} = \frac{1}{\sqrt{m \cdot \omega}}$, FORA's total expected running time is bounded by $O(\frac{\log(1/p_f)}{\epsilon \cdot \delta})$, which is a factor of $1/\epsilon$ smaller than that of *MC*.

To accelerate the *FP* phase, ResAcc [15] and SpeedPPR [16] have been proposed. The main idea of them is to reduce the total number of pushing operations. Specifically, ResAcc exploits the *looping phenomenon*, where some residues return back to a source node s. By accumulating returned residues, ResAcc avoids multiple pushing operations at s. SpeedPPR gradually reduces r_{max} so that only a node v with a larger $r(s,v)$ executes the pushing operation. Although both ResAcc and SpeedPPR have overcome the FORA's performance, they have not offered any positive solutions for accelerating the *MC* phase.

To our best knowledge, there are only two methods that have tried to optimize the *MC* phase. The first one is the extended version of FORA [21]. This method observes that α portion of random walks is expected to terminate without moving. By immediately recording the portion of such random walks, the total number of random walks can be reduced slightly. The second one is PAFO [22]. PAFO performs parallel random walks and simply counts the number of random walks that terminates at t. Since the update data array contains integer numbers, PAFO reduces the memory overhead for maintaining multiple copies between processors. Unfortunately, both methods are insufficient because the total number of obtaining neighbor nodes is still large.

IV. Proposed Method

In this section, we present the details of our proposed method. Our main idea stems from aggregating random walks at each node so that the total number of obtaining neighbor nodes can be minimized. We show that most of the random walks can be aggregated keeping theoretical guarantees because they do not need to memorize the starting node. Finally, we review the expected running time of the proposed method focusing on the total number of obtaining neighbor nodes.

Algorithm 2: Proposed Method

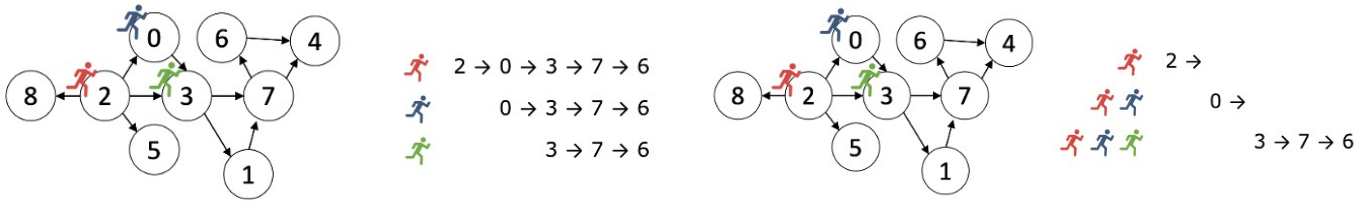
Input: Graph G, source node s, termination probability α, threshold r_{max}
Output: SSPPR score $\hat{\pi}(s,t)$ for all $t \in V$

1 invoke Forward Push with G, α, s and r_{max};
2 Let $\omega = \frac{(2\cdot\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\cdot\mu}$;
Independent Random Walk
3 IRW$(t) \leftarrow 0$ for all $t \in V$;
4 **for** *each* $t \in V$ *with* $\lfloor r(s,t) \cdot \omega \rfloor \geq 1$ **do**
5 Let IRW$(t) = \lfloor r(s,t) \cdot \omega \rfloor$;
6 **while** $\exists t \in V$ *such that* $IRW(t) > 0$ **do**
7 **for** $i = 1$ *to IRW*(t) **do**
8 Generate a random number r where $0 < r < 1$;
9 **if** $r < \alpha$ **then**
10 $\hat{\pi}(s,t) \leftarrow \hat{\pi}(s,t) + \frac{1}{\omega}$;
11 **else**
12 Choose u randomly where $u \in N_{out}(t)$;
13 IRW$(u) \leftarrow$ IRW$(u) + 1$;
14 IRW$(t) \leftarrow 0$;
Dependent Random Walk
15 **for** *each* $v \in V$ *with* $r(s,v) > 0$ **do**
16 Let $c_v = r(s,v) - \frac{\lfloor r(s,v)\cdot\omega \rfloor}{\omega}$;
17 Generate a random walk from v only once;
18 Let t be the last node of the random walk;
19 $\hat{\pi}(s,t) \leftarrow \hat{\pi}(s,t) + c_v$;
20 **return** $\hat{\pi}(s,t)$ for all $t \in V$;

A. Overview

Algorithm 2 shows the pseudo-code of the proposed method. The proposed method follows the two-stage computation: (1) Forward Push (*FP*), and (2) random walk Monte-Carlo simulation (*MC*). In the *FP* phase, we invoke the ordinary algorithm shown in Algorithm 1 (Line 1). In the *MC* phase, we perform two types of random walk, namely *Independent Random Walk (IRW)* (Lines 3-14), and *Dependent Random Walk (DRW)* (Lines 15-19). In IRW, each node v performs $\lfloor r(s,v)\cdot\omega \rfloor$ random walks (Lines 4-5). These random walks increase the reserve $\hat{\pi}(s,t)$ of a termination node t by $\frac{1}{\omega}$, which is a constant value (Lines 6-14). Since we do not have to memorize where each random walk is from, all random walks at the same node v can move by obtaining $N_{out}(v)$ only once. This realizes significant reduction of the total number of obtaining neighbor nodes. On the other hand, DRW performs sequential random walks as existing methods. Fortunately, the running cost of DRW is quite low because each node v performs DRW at most once. The details and theoretical guarantees of these two types of random walk are described in Section IV-B.

Besides, the expected running time of *MC* mainly depends on the total number of obtaining neighbor nodes. Since the proposed method can greatly reduce the total number of obtaining neighbor nodes, we can possibly establish a new



(a) Dependent Random Walk (DRW)

(b) Independent Random Walk (IRW)

Fig. 2: Assume that there are three random walks from node 2, 0, and 3 that have the same path $3 \rightarrow 7 \rightarrow 6$, the total number of obtaining neighbor nodes is reduced in IRW as shown in Fig 2(b) compared to DRW as shown in Fig 2(a).

expected running time. Motivated by this, we review the expected running time of the proposed method in Section IV-C.

B. Two Types of Random Walk

In the *MC* phase, a node v with $r(s,v) > 0$ performs random walks and increases the reserve $\hat{\pi}(s,t)$ of the termination node t. If this increment depends on the residue $r(s,v)$, we have to perform all random walks sequentially. In this case, the total number of obtaining neighbor nodes is quite large because each random walk needs to obtain neighbor nodes for every single step. We show an example in Figure 2(a). Assume that there are three random walks having similar paths, where $2 \rightarrow 0 \rightarrow 3 \rightarrow 7 \rightarrow 6$ (red), $0 \rightarrow 3 \rightarrow 7 \rightarrow 6$ (blue), $3 \rightarrow 7 \rightarrow 6$ (green). Even though all these random walks pass through node 3, 7, and 6 in this order, we have to obtain neighbor nodes of node 3, 7, and 6 three times, respectively.

To tackle this problem, we perform random walks that depend on $r(s,v)$ as less as possible. Our idea is to unify the increment value of $\hat{\pi}(s,t)$ during IRW so that all random walks at the same node v can move by obtaining $N_{out}(v)$ only once. We show an example of IRW in Figure 2(b). The three random walks shown in Fig 2(a) can reach the termination node 6 at the same time. Since we suffice to obtain neighbor nodes at node 3, 7, and 6 only once for each, we can reduce the total number of obtaining neighbor nodes. Therefore, we can rapidly complete IRW. Although we still have to sequentially perform DRW, whose the increment value depends on $r(s,v)$, it is guaranteed that running cost of DRW is quite low because every node v needs to perform DRW at most once.

We show that the proposed method returns $\hat{\pi}(s,t)$ for all $t \in V$ satisfying the Definition 1. We first introduce the following equation [22]:

$$\mathbb{E}\left[\sum_{j=1}^{\lfloor r(s,v)\cdot\omega \rfloor} \frac{1}{\omega} \cdot X_j + c_v \cdot X_j\right] = r(s,v)\cdot\pi(v,t), \quad (4)$$

where X_j denotes a Bernoulli variable that takes value 1 if the j-th random walk terminates at t, and value 0 otherwise. Observe that $\sum_{j=1}^{\lfloor r(s,v)\cdot\omega \rfloor} \frac{1}{\omega} \cdot X_j$ is exactly the amount of increment that $\hat{\pi}(s,t)$ receives from a node v during IRW, and $c_v \cdot X_j$ is that of during DRW. The total increment of $\hat{\pi}(s,t)$ follows

$$\mathbb{E}\left[\sum_{v\in V} r(s,v)\cdot\pi(v,t)\right] = \sum_{v\in V} r(s,v)\cdot\pi(v,t).$$

By combining Equation (3) and the Chernoff bound [23], the proposed method satisfies the Definition 1.

It is worth mentioning that the proposed method has two improvements over [22]. First, we apply the advantage of Equation (4), where we do not have to memorize the starting node of each random walk, to every passed node while authors in [22] only consider the termination node. Second, the proposed method requires no preprocessings while authors in [22] have to execute heavy preprocessings and hold them as indices.

C. Reviewing Expected Running Time

It has been proved that the expected running time of *FP* shown in Algorithm 1 can be bounded by $O\left(\frac{1}{\alpha\cdot r_{max}}\right)$ [20]. Since the proposed method uses Algorithm 1 in the *FP* phase, we can also define the expected running time of *FP* in the proposed method as $O\left(\frac{1}{\alpha\cdot r_{max}}\right)$.

Let ω_{sum} be the total number of random walks, existing methods expect that *MC* has $O\left(\frac{\omega_{sum}}{\alpha}\right)$ running time. This is because every random walk is performed sequentially, and one random walk moves $\frac{1}{\alpha}$ steps on average. Obviously, the expected running time of *MC* is determined by the total number of obtaining neighbor nodes. Therefore, we estimate this number to define the expected running time of *MC* in the proposed method.

We first exploit the expected running time of DRW. In DRW, each node v with $r(s,v) > 0$ performs a random walk only once. Therefore, we easily know that there are at most n random walks. Since these random walks are performed sequentially, we can define the total running time of DRW as $O\left(\frac{n}{\alpha}\right)$.

Next, we exploit the expected running time of IRW. For the ease of analysis, we first define the iteration of IRW. We denote $I^{(k)}$ as the set of all active nodes at the beginning of the k-th iteration, where a node t is active if IRW$(t) \geq 1$. Initially, $I^{(0)}$ contains every node t with $\lfloor r(s,t)\cdot\omega \rfloor \geq 1$. $I^{(k)}$ is the set of every node t with IRW$(t) \geq 1$ after the $(k-1)$-th iteration. In addition, we denote $\omega_{sum}^{(k)}$ as the remaining number of random

Name	n	m	m/n	Type
DBLP	317K	2.10M	6.62	Undirected
Pokec	1.63M	30.6M	18.8	Directed
LiveJournal	4.85M	68.4M	14.1	Directed
Orkut	3.07M	234M	76.3	Undirected

TABLE II: Datasets description

walks at the beginning of the k-th iteration. Based on these definitions, we prove the following theorem.

Theorem 1. *The expected running time of IRW is* $O\left(n \cdot \log(m \cdot r_{max} \cdot \omega)\right)$.

Proof. Consider the $(k+1)$-th iteration, we have

$$\omega_{sum}^{(k+1)} \leq (1-\alpha) \cdot \omega_{sum}^{(k)}.$$

Based on this, we can easily get

$$\omega_{sum}^{(k)} \leq (1-\alpha)^k \cdot \omega_{sum}^{(0)}. \tag{5}$$

Besides, $\omega_{sum}^{(0)}$ follows

$$\begin{aligned} \omega_{sum}^{(0)} = \sum_{t \in V} \lfloor r(s,t) \cdot \omega \rfloor &\leq \sum_{t \in V} \lfloor r_{max} \cdot d_{out}(t) \cdot \omega \rfloor \\ &\leq m \cdot r_{max} \cdot \omega. \end{aligned} \tag{6}$$

We assume that IRW finishes when the value of $\omega_{sum}^{(k)}$ becomes smaller than one. By combining Equation (5) and Equation (6), we know that IRW needs $K = O\left(\log(m \cdot r_{max} \cdot \omega)\right)$ iterations to satisfy the condition $\omega_{sum}^{(k)} < 1$.

In the k-th iteration, $\left|I^{(k)}\right|$ numbers of nodes are still active. Observe that $\left|I^{(k)}\right| \leq n$, the total number of obtaining neighbor nodes follows

$$\sum_{k=0}^{K-1} \left|I^{(k)}\right| \leq \sum_{k=0}^{K-1} n = n \cdot K.$$

Therefore, the expected running time of IRW is $O\left(n \cdot \log(m \cdot r_{max} \cdot \omega)\right)$. This finishes the proof. □

V. Evaluation

A. Experimental Setup

We conducted all experiments on a Linux 20.04 server with dual Intel Xeon E5-2643 processors and 94GiB memory. All algorithms were implemented with C++ and compiled with G++ 9.4.0 using the -O3 optimization.

Real-World Graphs: We used four real-world graphs, DBLP collaboration network (DBLP) [24], Pokec social network (Pokec) [17], LiveJournal social network (LiveJournal) [25] and Orkut social network (Orkut) [24]. All the graphs are available on the Stanford SNAP library[1]. Details of all graphs are shown in Table II.

[1]https://snap.stanford.edu/data/

Experiments: We compared the proposed method with three index-free methods: FORA [14], ResAcc [15] and SpeedPPR [16]. The computational efficiency was measured through the overall running time. We generated 50 query source nodes uniformly at random, and report the average results of these 50 nodes. To generate random numbers efficiently, we used the latest version of SIMD-oriented Fast Mersenne Twister Library[2]. The source codes we used for the evaluation can be found online[3].

Parameter Settings: Following the existing methods, we set $\alpha = 0.2$, $\delta = 1/n$, $p_f = 1/n$ and $\epsilon = 0.5$. By default, we set $r_{max} = \frac{1}{\sqrt{m \cdot \omega}}$, where $\omega = \frac{(2 \cdot \epsilon/3+2) \cdot \log(2/p_f)}{\epsilon^2 \cdot \delta}$, following the existing methods [14], [15]. As for SpeedPRR, we set $r_{max} = \frac{1}{\omega}$ according to the original paper [16]. Moreover, there are two additional parameters with ResAcc. We set these parameters to realize the best performance.

B. Experimental Results

Figure 3 shows the overall running time in log scale. The running time of *Proposed* was smaller than the existing methods on all datasets except DBLP. On DBLP, *Proposed* was slightly worse than SpeedPPR. This is because differences in computational efficiency rarely appear with small graphs. *Proposed* realized significant speedup, and the maximum speedup of *Proposed* was 3.3x, 3.3x, and 2.1x compared to FORA, ResAcc, and SpeedPPR, respectively. On Orkut, *Proposed* was over twenty seconds faster than SpeedPPR, which is the fastest existing method. This shows a considerable superiority of *Proposed* in terms of computational efficiency. It is worth pointing out that *Proposed* was more effective on larger graphs. Therefore, we estimate that the proposed method can greatly overcome existing methods on billion-scale graphs, which are emerging in real-world applications.

VI. Conclusion

In this paper, we proposed a node-centric random walk to accelerate random walk Monte-Carlo simulation. Our main idea stems from aggregating random walks at each node so that the total number of obtaining neighbor nodes can be minimized. The proposed method can aggregate most of the random walks because they do not need to memorize the starting node. Moreover, we reviewed the expected running time of the proposed method. By focusing on the total number of obtaining neighbor nodes, we established a new expected running time. Extensive experiments using real-world graphs showed that the proposed method is up to 3.3x faster than the existing methods.

As future work, we will concentrate on further improving the performance of both Forward Push and random walk Monte-Carlo simulation.

[2]http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/SFMT/

[3]https://github.com/StyLishPoor/Accelerated-Random-Walk

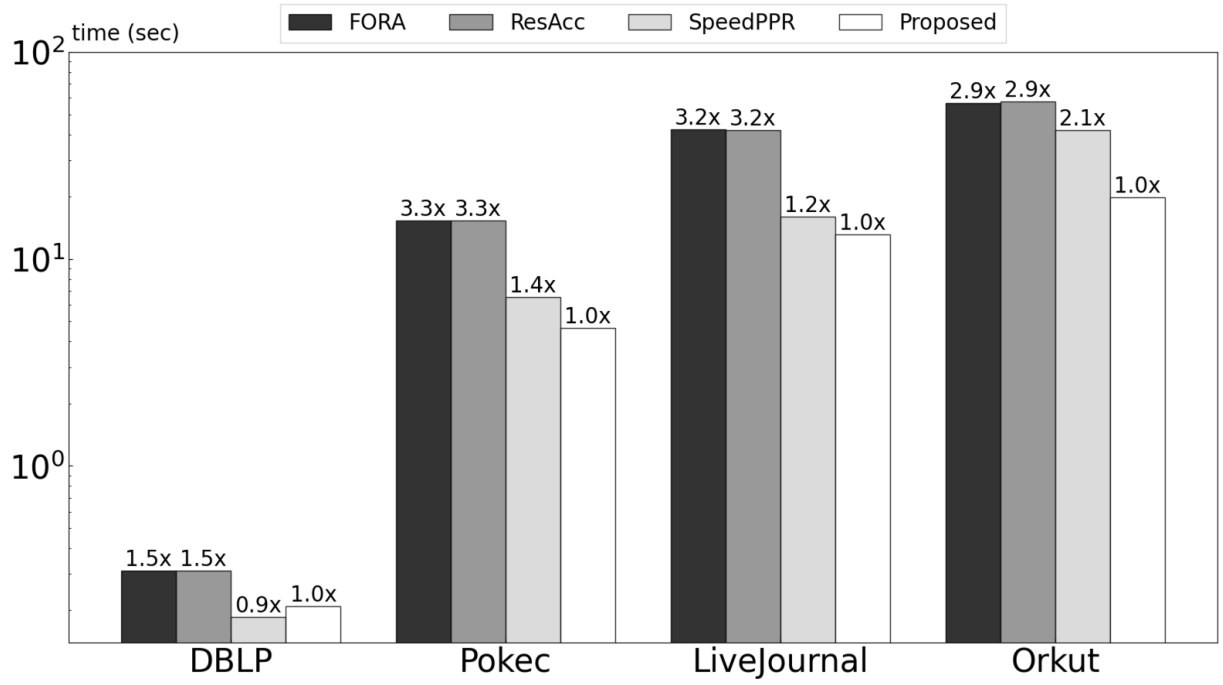


Fig. 3: The overall running time on each dataset. The number **d.d**x over each bar means that Proposed is **d.d**x faster than the others.

REFERENCES

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

[2] R. Andersen, C. Borgs, J. Chayes, J. Hopcroft, K. Jain, V. Mirrokni, and S. Teng, "Robust pagerank and locally computable spam detection features," in *Proc. AIRWeb*, 2008, pp. 69–76.

[3] L. Backstrom and J. Leskovec, "Supervised random walks: predicting and recommending links in social networks," in *Proc. ACM WSDM*, 2011, pp. 635–644.

[4] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *Proc. ACM WWW*, 2013, pp. 505–514.

[5] Y. Gao, X. Yu, and H. Zhang, "Overlapping community detection by constrained personalized pagerank," *Expert Systems with Applications*, vol. 173, p. 114682, 2021.

[6] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proc. ACM SIGKDD*, 2017, pp. 555–564.

[7] J. J. Whang, D. F. Gleich, and I. S. Dhillon, "Overlapping community detection using neighborhood-inflated seed expansion," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1272–1284, 2016.

[8] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *Proc. ICML*. PMLR, 2018, pp. 5453–5462.

[9] J. Klicpera, A. Bojchevski, and S. Günnemann, "Predict then propagate: Graph neural networks meet personalized pagerank," in *Proc. ICLR*, 2019.

[10] A. Bojchevski, J. Klicpera, B. Perozzi, A. Kapoor, M. Blais, B. Rózemberczki, M. Lukasik, and S. Günnemann, "Scaling graph neural networks with approximate pagerank," in *Proc. ACM SIGKDD*, 2020, pp. 2464–2473.

[11] J. Jung, N. Park, S. Lee, and U. Kang, "Bepi: Fast and memory-efficient method for billion-scale random walk with restart," in *Proc. ACM SIGMOD*, 2017, pp. 789–804.

[12] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li, "Hubppr: effective indexing for approximate personalized pagerank," *Proc. VLDB Endowment*, vol. 10, no. 3, pp. 205–216, 2016.

[13] M. Yoon, J. Jung, and U. Kang, "Tpa: Fast, scalable, and accurate method for approximate random walk with restart on billion scale graphs," in *Proc. IEEE ICDE*, 2018, pp. 1132–1143.

[14] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang, "Fora: simple and effective approximate single-source personalized pagerank," in *Proc. ACM SIGKDD*, 2017, pp. 505–514.

[15] D. Lin, R. C.-W. Wong, M. Xie, and V. J. Wei, "Index-free approach with theoretical guarantee for efficient random walk with restart query," in *Proc. IEEE ICDE*, 2020, pp. 913–924.

[16] H. Wu, J. Gan, Z. Wei, and R. Zhang, "Unifying the global and local approaches: an efficient power iteration with forward push," in *Proc. ACM SIGMOD*, 2021, pp. 1996–2008.

[17] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International scientific conference and international workshop present day trends of innovations*, vol. 1, no. 6. Present Day Trends of Innovations Lamza Poland, 2012.

[18] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós, "Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments," *Internet Mathematics*, vol. 2, no. 3, pp. 333–358, 2005.

[19] G. Jeh and J. Widom, "Scaling personalized web search," in *Proc. ACM WWW*, 2003, pp. 271–279.

[20] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *Proc. IEEE FOCS*, 2006, pp. 475–486.

[21] S. Wang, R. Yang, R. Wang, X. Xiao, Z. Wei, W. Lin, Y. Yang, and N. Tang, "Efficient algorithms for approximate single-source personalized pagerank queries," *ACM Transactions on Database Systems*, vol. 44, no. 4, pp. 1–37, 2019.

[22] R. Wang, S. Wang, and X. Zhou, "Parallelizing approximate single-source personalized pagerank queries on shared memory," *The VLDB Journal*, vol. 28, no. 6, pp. 923–940, 2019.

[23] F. Chung and L. Lu, "Concentration inequalities and martingale inequalities: a survey," *Internet mathematics*, vol. 3, no. 1, pp. 79–127, 2006.

[24] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[25] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proc. ACM SIGKDD*, 2006, pp. 44–54.