Received 15 December 2022, accepted 9 January 2023, date of publication 16 January 2023, date of current version 24 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3237738

RESEARCH ARTICLE

Cache-Efficient Approach for Index-Free Personalized PageRank

KOHEI TSUCHIDA[1], NAOKI MATSUMOTO[2], ANDREW SHIN[2], AND KUNITAKE KANEKO[2,3], (Member, IEEE)
[1]Graduate School of Science and Technology, Keio University, Kanagawa 223-8522, Japan
[2]Research Institute for Digital Media and Content, Keio University, Kanagawa 223-8523, Japan
[3]Faculty of Science and Technology, Keio University, Kanagawa 223-8522, Japan

Corresponding author: Kohei Tsuchida (nora@inl.ics.keio.ac.jp)

ABSTRACT *Personalized PageRank (PPR)* measures the importance of vertices with respect to a source vertex. Since real-world graphs are evolving rapidly, PPR computation methods need to be index-free and fast. Unfortunately, existing index-free methods suffer from cache misses. They follow the state-of-the-art algorithm that first performs the Forward Push (FP) phase and subsequently runs the random walk Monte-Carlo simulation (MC) phase. Although existing methods succeed in reducing cache misses in the FP phase, an inefficient data layout limits their performance improvement. Besides, existing methods have overlooked the importance of reducing cache misses in the MC phase. In this paper, we propose a cache-efficient approach that accelerates both FP and MC phases. In the FP phase, we first reorder the data layout with low overheads. Specifically, we utilize the Breadth First Search result so that vertices near the source vertex are co-located on the reordered data layout. We subsequently perform optimized FP, namely *Distance-Extension Forward Push (DEFP)*. By preferentially proceeding FP around the source vertex, DEFP improves memory access locality. In the MC phase, we perform optimized MC, namely *Vertex-Centric Random Walk (VCRW)*. VCRW aggregates random walks at each vertex to eliminate redundant memory access for repeatedly obtaining neighbor vertices. We prove that most of the random walks can be aggregated while maintaining accuracy guarantees. Experimental results show that the proposed method is up to 4.7x faster than existing index-free methods and outperforms the state-of-the-art index-oriented method under rigorous accuracy guarantees.

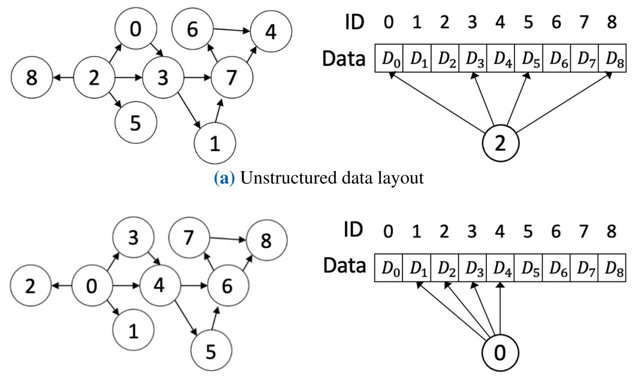
INDEX TERMS Personalized pagerank, index-free, graph reordering, forward push, random walk monte-carlo simulation.

I. INTRODUCTION

Personalized PageRank (PPR) [1] is one of the most popular graph computations to measure the proximity of vertices. Of our particular interest is the single-source PPR (SSPPR) among several PPR variants, such as single-target PPR, pair-wise PPR, and fully PPR. Given a graph $G = (V, E)$, a termination probability α, a source vertex s, and a target vertex t, the SSPPR score $\pi(s, t)$ is defined as the probability that an α-decay random walk starting from s terminates at t in G. When performing an α-decay random walk, a random walker starting from s terminates at the current vertex with probability α or moves to a neighbor vertex with probability $1 - \alpha$. By this definition, the SSPPR scores can also be regarded as the measure of relative importance of all vertices with respect to s. Based on this, SSPPR has various real-world applications, such as spam detection [2], link prediction [3], social recommendation [4], community detection [5], [6], [7], graph learning [8], [9], [10], and so on.

Nowadays, massive real-world graphs are evolving rapidly. In this scenario, index-oriented SSPPR computation methods [11], [12], [13] are impractical because they need to hold huge indices and update them frequently. Therefore, SSPPR computation methods need to be index-free and fast.

The associate editor coordinating the review of this manuscript and approving it for publication was Chong Leong Gan.



(a) Unstructured data layout

(b) Reordered data layout

FIGURE 1. Suppose that vertex 2 in Fig. 1(a) and vertex 0 in Fig. 1(b) access neighbor vertices' data. Even though the two vertices have identical positions in each graph, memory access locality in Fig. 1(b) is better than that in Fig. 1(a).

The state-of-the-art index-free methods [14], [15] follow the FORA algorithm [16] that first performs the Forward Push (FP) phase and subsequently runs the random walk Monte-Carlo simulation (MC) phase. Although these methods have steadily accelerated SSPPR computation, they still suffer from a large number of cache misses in both FP and MC phases.

In the FP phase, irregular memory access primarily causes frequent cache misses. As shown in Fig. 1(a), a real-world graph tends to have an unstructured data layout. Assuming that vertex 2 accesses its neighbor vertices, irregular memory access occurs because these neighbor vertices are not co-located on memory. On an unstructured data layout, every vertex repeatedly accesses neighbor vertices until the FP phase finishes, which leads to a significant number of cache misses in total. Since existing methods [14], [15] optimize the FP phase without considering the data layout, their performance improvement is insufficient. Graph reordering [17], [18], [19], [20], [21], [22] is widely used to optimize the data layout by relabeling vertex IDs. As shown in Fig. 1(b), a reordered data layout improves memory access locality. In general, reordering methods assign close IDs to frequently accessed vertices so that processors can reuse cached data. Although existing reordering methods can accelerate various graph computations, their time-consuming procedures to find an optimal relabeling cause an end-to-end slowdown. Therefore, to alleviate irregular memory access, we need to conduct lightweight reordering that captures the memory access pattern of the FP phase.

In the MC phase, cache misses are mainly caused by redundant memory access. For accuracy guarantees, existing methods perform a large number of α-decay random walks sequentially to memorize the starting vertex of each random walk. Through this sequential process, each random walk needs to obtain neighbor vertices for every single step to decide the next destination. Therefore, we need to obtain neighbor vertices multiple times at each vertex, which leads to redundant memory access. Notably, existing index-free methods have only focused on optimizing the FP phase, and they overlook this redundant memory access. To overcome this problem, we need to reduce the total number of operations to obtain neighbor vertices at each vertex.

In this paper, we propose a cache-efficient approach that significantly accelerates the FORA algorithm. To reduce cache misses, we focus on optimizing the computational procedure of both FP and MC phases. While some methods [14], [23] accelerate the FORA algorithm by modifying the timing at which the FP phase switches to the MC phase, examining an appropriate switching timing lies outside the scope of this paper. To optimize the FP phase, we first conduct lightweight reordering. We observe that vertices near the source vertex are frequently accessed during the FP phase. Therefore, the FP phase can be accelerated if vertices near the source vertex have close IDs. To realize this, we reorder the data layout according to the Breadth First Search (BFS) result from the source vertex. This reordering can preferentially assign close IDs to vertices near the source vertex with low overheads. To fully utilize the reordering result, we introduce the optimized FP, namely *Distance-Extension Forward Push (DEFP)*. DEFP executes FP within k distances from the source vertex and gradually increases the k value, which can reduce the total number of operations during the FP phase. In the MC phase, we perform *Vertex-Centric Random Walk (VCRW)*. VCRW aggregates random walks at each vertex to reduce the total number of operations to obtain neighbor vertices. This aggregation eliminates redundant memory access for repeatedly obtaining neighbor vertices at each vertex. We prove that VCRW can aggregate most of the random walks while maintaining accuracy guarantees.

Our contributions are summarized as follows:

- We propose a cache-efficient approach for fast index-free PPR computation. In the FP phase, the proposed method first conducts lightweight reordering according to the Breadth First Search result. On the reordered data layout, the proposed method performs *Distance-Extension Forward Push*. In the MC phase, the proposed method performs *Vertex-Centric Random Walk*. These techniques can significantly reduce cache misses.
- We conduct extensive experiments using six real-world graphs. Experimental results show that the proposed method is up to 4.7× faster than the existing methods. We confirm that the proposed method reduces cache misses on both the L1 cache and L3 cache. Notably, the proposed method outperforms the state-of-the-art index-oriented method under rigorous accuracy guarantees.

This paper extends our previous work [24] that originally proposed the idea of aggregating random walks. While [24] measures the computational efficiency simply through the running time, this paper further investigates the cache performance, which is our major interest, in addition to the running time measurement. Additionally, this paper has the following three novelties. First, we focus on optimizing the FP phase by graph reordering and extend the survey to include graph reordering methods. Second, we theoretically prove that the proposed method guarantees the accuracy. Third, we conduct far more extensive experiments considering various applications scenarios.

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 the proposed method. We evaluate the proposed method through extensive experiments using real-world graphs in Section V. The source code of the proposed method can be found online.[1] 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 vertices V and a set of edges E. $n = |V|$, $m = |E|$ are the number of vertices 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 the set of in-neighbor vertices (resp. out-neighbor vertices) of $v \in V$, and let $d_{in}(v)$ (resp. $d_{out}(v)$) denote the in-degree (resp. out-degree) of $v \in V$. Given a source vertex s, the SSPPR score $\pi(s, t)$ of t is defined as the probability that an α-decay random walk starting from s terminates at t in G. An α-decay random walk terminates at the current vertex with α probability or moves to an out-neighbor vertex with $1 - \alpha$ probability. In addition, let $dist(s, v)$ denote the shortest distance from s to v in G, $V_k = \{v \in V \mid dist(s, v) = k\}$ denote the set of vertices whose $dist(s, v) = k$, and $U_k = \{v \in V \mid dist(s, v) \leq k\}$ denote the set of vertices whose $dist(s, v) \leq k$. In this paper, we focus on the Approximate SSPPR query, hereafter referred to simply as PPR (Definition 1). Table 1 summarizes the notations we frequently use in this paper.

Definition 1: (PPR Query) Given a graph $G = (V, E)$, a source vertex s, a threshold δ, an error bound ϵ, and a failure probability p_f, PPR query returns the estimated PPR 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.

Graph format: We represent a graph G by the well-known *Compressed Sparse Row (CSR)* format [25]. CSR uses two arrays, namely *Coordinate Array* (CA) and *Offset Array* (OA). The CA contiguously stores the neighbor vertices of each vertex $v \in V$. The OA stores each vertex's starting offset in the CA. For instance, to obtain neighbor vertices of a vertex v, a program accesses from the OA[v]-th entry to the OA[$v + 1$]-th entry on the CA. A directed graph requires two CSR representations for out-neighbor vertices and in-neighbor vertices, respectively. We show an example of CSR representation for out-neighbor vertices in Fig. 2. Note that in-neighbor vertices can be represented in the same way.

[1] https://github.com/StyLishPoor/Cache-Efficient-PPR

TABLE 1. Frequently used notations.

Notation	Description
G	The input graph
V	The vertex set of G
E	The edge set of G
n	The number of vertices in G
m	The number of edges in G
$N_{in}(v)$	The in-neighbor vertices of a vertex v
$N_{out}(v)$	The out-neighbor vertices of a vertex v
$d_{in}(v)$	The in-degree of a vertex v
$d_{out}(v)$	The out-degree of a vertex v
s	The source vertex of PPR queries
$dist(s, v)$	The shortest distance from s to v in G
V_k	The set of vertices whose $dist(s, v) = k$
U_k	The set of vertices whose $dist(s, v) \leq k$
$\pi(s, v)$	The PPR score of a vertex v
$r(s, v)$	The residue of a vertex v
$\hat{\pi}(s, v)$	The reserve of a vertex v
r_{max}	The residue threshold
r_{sum}	The sum of all vertices' residues
α	The random walk termination probability
δ, ϵ, p_f	The parameters of PPR queries

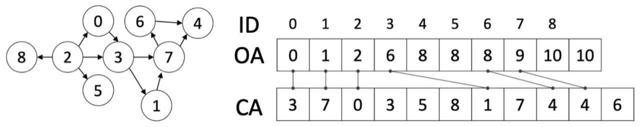


FIGURE 2. The CSR representation for out-neighbor vertices in a toy directed graph.

Algorithm 1: Forward Push

```
Input: Graph $G$, source vertex $s$, termination probability
       $\alpha$, 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
       # Pushing operation
4      $\hat{\pi}(s, t) \leftarrow \hat{\pi}(s, t) + \alpha \cdot r(s, t)$;
5      if $d_{out}(t) \neq 0$ then
6          for each $u \in N_{out}(t)$ do
7              $r(s, u) \leftarrow r(s, u) + (1 - \alpha) \cdot \frac{r(s,t)}{d_{out}(t)}$;
8      else
9          $r(s, s) \leftarrow r(s, s) + (1 - \alpha) \cdot r(s, t)$;
10     $r(s, t) \leftarrow 0$;
11 return $r(s, t)$ and $\hat{\pi}(s, t)$ for all $t \in V$;
```

B. FORWARD PUSH

Forward Push (FP) [26] is a local update method. FP simulates random walks in a deterministic way by repeatedly pushing the probability mass to out-neighbor vertices. Algorithm 1 shows the pseudo-code of FP. FP maintains residue $r(s, t)$ and reserve $\hat{\pi}(s,t)$ for each vertex $t \in V$. At the beginning, FP initializes $r(s,t)$ and $\hat{\pi}(s,t)$ (Lines 1-2). A vertex t becomes active when $r(s,t) > r_{max} \cdot d_{out}(t)$. An active vertex t executes *pushing operation* that increases $\hat{\pi}(s,t)$ by converting α portion of $r(s,t)$ and transfers $1-\alpha$ portion of $r(s,t)$ to $N_{out}(t)$ (Lines 4-7). If t is a dangling vertex, t transfers $1-\alpha \cdot r(s,t)$ to s (Lines 8-9). t finishes the pushing operation after resetting $r(s,t)$ to zero (Line 10). When there are no active vertices, $r(s,t)$ and $\hat{\pi}(s,t)$ are returned as final residue and the PPR score, respectively (Line 11). The expected running time of FP is $O\left(\frac{1}{\alpha \cdot r_{max}}\right)$. However, FP cannot provide any accuracy guarantees.

C. RANDOM WALK MONTE-CARLO SIMULATION

Random walk Monte-Carlo simulation (MC) is a classic and straightforward solution to answer PPR queries [27], [28]. MC performs ω random walks from s, and measures the fraction of random walks that 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 [27]. Assuming that a given graph is scale-free, where the number of edges is $m = O(n \cdot \log n)$, the expected running time is bounded by $O\left(\frac{\log(1/p_f)}{\epsilon^2\cdot\delta}\right)$, which is infeasible with massive real-world graphs.

III. RELATED WORK

A. PERSONALIZED PAGERANK

FORA [16] is the first method that combines FP and MC. FORA first invokes FP with early termination and subsequently performs random walks to obtain accuracy guarantees. FORA utilizes the following invariant [26]:

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

where $\pi(s,t)^{\circ}$ is the reserve of a vertex t after the FP phase. Since computing $\pi(v,t)$ for all $v \in V$ 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 vertex v performs $\lceil r(s,v)\cdot\omega \rceil$ random walks, where $\omega = \frac{(2\cdot\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\cdot\delta}$. Recall that the expected running time of FP is $O\left(\frac{1}{\alpha\cdot r_{max}}\right)$ and MC in FORA has at most $O\left(\frac{m\cdot r_{max}\cdot\omega}{\alpha}\right)$ running time. By setting $r_{max} = \frac{1}{\sqrt{m\cdot\omega}}$, the total expected running time of FORA is bounded by $O\left(\frac{\log(1/p_f)}{\epsilon\cdot\delta}\right)$, which is a factor of $1/\epsilon$ smaller than that of MC.

To optimize the computational procedure of the FP phase, ResAcc [14] and SpeedPPR [15] have been proposed. Their main idea is to reduce the total number of pushing operations. Specifically, ResAcc exploits the *looping phenomenon*, where some residues return back to s. By accumulating returned residues, ResAcc avoids multiple pushing operations at s. SpeedPPR gradually reduces r_{max} so that vertices holding a large residue execute the pushing operation. Although both ResAcc and SpeedPPR have outperformed FORA, they have overlooked the importance of reordering the data layout and optimizing the computational procedure of the MC phase.

Index-oriented methods [11], [12], [23], [29] aim to answer PPR queries rapidly by using precomputed results. Matrix-based methods [11], [29] convert the adjacent matrix so that the converted matrix has a large and easy-to-invert submatrix. Since these methods hold the converted matrix as indices, they cause huge space overheads. HubPPR [12] executes random walks from high-degree vertices and stores the results as indices. The indices are combined with Backward search [30]. FORA+ [23] is the state-of-the-art method. FORA+ samples random walk from all the vertices in advance and uses them directly in the MC phase. Despite the efficiency of query responses, index-oriented methods are impractical due to high overheads of updating indices.

Other methods answer PPR queries with particular settings. Depending on the query types, there are three lines of research: top-k query [31], [32], batch one-hop query [33], and setting small α [34]. Several methods focus on dynamic graphs [35], [36], [37] or weighted graphs [38]. In addition, there are a number of methods considering distributed environments [39], [40], [41] and parallel computation [42].

B. GRAPH REORDERING

Connectivity-based methods [18], [19] assign close vertex IDs to densely connected vertices. Rabbit-Order [19] reorders the data layout based on the community structure. Since communities consist of densely interconnected vertices, Rabbit-Order assigns consecutive IDs to vertices in each community. Gorder [18] is the state-of-the-art method that offers significant performance improvement. Gorder assigns consecutive IDs to vertices sharing a large number of common vertices. Specifically, let R_w be w vertices that are recently relabeled with a new ID. A vertex v obtains a new ID if v shares the largest number of common neighbor vertices with R_w. Although connectivity-based methods can accelerate various graph computations, time-consuming reordering procedures in these methods cause an end-to-end slowdown.

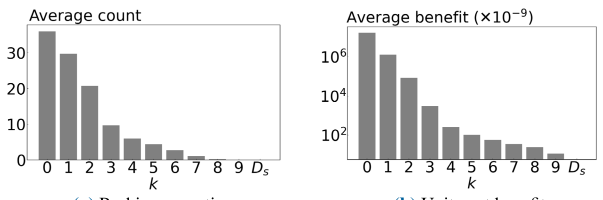
To alleviate huge reordering overheads, degree-based methods [20], [21], [22] have attracted much attention in the literature. In general, degree-based methods assign close IDs to high-degree vertices because these vertices tend to be frequently accessed in graph computations. Since degree-based methods simply use the degree information of vertices, they can reduce the reordering time. However, it remains challenging to realize enough performance improvement comparable with connectivity-based methods.

IV. PROPOSED METHOD

In this section, we present the proposed method. We first outline the overall approach in Section IV-A. The details of the proposed method are described in Section IV-B and

Algorithm 2: the Proposed Method

```
Input: Graph $G$, source vertex $s$, termination probability
       $\alpha$, threshold $r_{max}$
Output: PPR score $\hat{\pi}(s,t)$ for all $t \in V$
1 $G^* \leftarrow$ BFS-based lightweight reordering with $G$, $s$;
2 invoke Distance-Extension Forwrd Push with $G^*$;
3 perform Vertex-Centric Random Walk with $G^*$;
4 return $\hat{\pi}(s,t)$ for all $t \in V$;
```



(a) Pushing operation (b) Unit-cost benefit

FIGURE 3. Analyzing the workload of V_k during the FP phase on Pokec dataset [43].

Section IV-C. Finally, we elaborate the running time analysis in Section IV-D.

A. OVERVIEW

Algorithm 2 shows the pseudo-code of the proposed method. To reduce cache misses, the proposed method optimizes the computational procedure of both FP and MC phases. In the FP phase, the proposed method first conducts lightweight BFS-based reordering (Line 1). This reordering preferentially assigns close vertex IDs to vertices with small $dist(s,v)$ because these vertices perform a large number of pushing operations. On the reordered data layout, the proposed method invokes *Distance-Extension Forward Push (DEFP)* (Line 2). In the MC phase, the proposed method performs *Vertex-Centric Random Walk (VCRW)* (Line 3). VCRW aggregates random walks at each vertex to reduce the total number of operations to obtain neighbor vertices. Finally, the proposed method returns the PPR score $\hat{\pi}(s,t)$ satisfying Definition 1 for all $t \in V$ (Line 4).

B. DISTANCE-EXTENSION FORWARD PUSH

We first analyze the workload of the FP algorithm shown in Algorithm 1 focusing on $dist(s,v)$. Let D_s denote the maximum distance from s in G. We selected one source vertex, and then measured the average number of pushing operations and the average *unit-cost benefit* [15] of V_k. Note that *unit-cost benefit* of the pushing operation on v is defined as $\frac{\alpha \cdot r(s,v)}{d_{out}(v)}$ because v needs to access $d_{out}(v)$ vertices to reduce the sum of residues r_{sum} by $\alpha \cdot r(s,v)$. We conducted the same analysis on 50 source vertices selected uniformly at random, and we confirmed that all vertices indicate a similar tendency. Therefore, we report the results from source vertex s whose D_s is closest to the average D_s computed from 50 source vertices. Fig. 3 shows the corresponding results. We can see

Algorithm 3: BFS-Based Lightweight reordering

```
Input: Graph $G$, source vertex $s$
Output: Reordered Graph $G^*$, $DistanceIdx$
1  $Found[t] \leftarrow false$ for all $t \in V$;
2  $Order[t] \leftarrow empty$ for all $t \in V$;
3  $DistanceIdx[d] \leftarrow empty$ for $0 \le d \le D_s$;
4  $Order[0] \leftarrow s$, $DistanceIdx[0] \leftarrow 0$;
5  $sentinel \leftarrow 1$;
6  $distance \leftarrow 1$;
7  $left \leftarrow 0$, $right \leftarrow 1$;
8  while $left < right$ do
9      if $left == sentinel$ then
10         $DistanceIdx[distance] \leftarrow right$;
11         $distance \leftarrow distance + 1$;
12         $sentinel \leftarrow right$;
13     $v \leftarrow Order[left]$;
14     for each $u \in N_{out}(v)$ do
15         if $Found[u] == false$ then
16             $Found[u] \leftarrow true$;
17             $Order[right] \leftarrow u$;
18             $right \leftarrow right + 1$;
19     $left \leftarrow left + 1$;
   # Relabel the vertex IDs
20 for $new\_id = 0$ to $n$ do
21     convert $Order[new\_id]$ in $G$ to $new\_id$ in $G^*$;
22 return $G^*$, $DistanceIdx$;
```

that vertices near s tend to perform a large number of pushing operations with large benefits.

Based on this, the proposed method reorders the data layout to improve memory access locality of V_k with small k. Moreover, the proposed method establishes the *Distance-Extension* strategy that performs pushing operations with U_k and gradually increases the k value. Since pushing operations on V_k with small k have large benefits, our strategy can efficiently proceed the FP phase.

Algorithm 3 shows the pseudo-code of the reordering procedure. We perform BFS from s and assign a new ID in ascending order to a found vertex. Considering that a vertex $v \in V_k$ with small k expands the searching area, the majority of $N_{out}(v)$ have not been found yet. Therefore, most of $N_{out}(v)$ can obtain consecutive vertex IDs. As a result, vertices in V_{k+1} have close IDs because V_{k+1} consists of out-neighbor vertices of V_k. To quickly determine U_k, we also calculate *DistanceIdx* along with the BFS procedure, where $DistanceIdx[k]$ denotes the maximum vertex ID of V_k. Considering that BFS has $O(m+n) = O\left(\sum_{v \in V}(d_{out}(v)+1)\right)$ running time, Algorithm 3 is fairly lightweight compared with the representative reordering method Gorder [18] that has $O\left(\sum_{v \in V}(d_{out}(v))^2\right)$ running time.

Algorithm 4 shows the pseudo-code of DEFP. As mentioned before, DEFP performs FP with U_k and iteratively

```
Algorithm 4: Distance-Extension Forward Push
Input: Graph $G^*$, source vertex $s$, termination
       probability $\alpha$, threshold $r_{max}$, $DistanceIdx$
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  for $k = 0$ to $D_s$ do
4      for $t = 0$ to $DistanceIdx[k]$ do
5          if $r(s,t) > d_{out}(t) \cdot r_{max}$ then
6              $\hat{\pi}(s,t) \leftarrow \hat{\pi}(s,t) + \alpha \cdot r(s,t)$;
7              if $d_{out}(t) \neq 0$ then
8                  for each $u \in N_{out}(t)$ do
9                      $r(s,u) \leftarrow r(s,u) + (1-\alpha) \cdot \frac{r(s,t)}{d_{out}(t)}$;
10             else
11                 $r(s,s) \leftarrow r(s,s) + (1-\alpha) \cdot r(s,t)$;
12             $r(s,t) \leftarrow 0$;
13 while $\exists t \in V$ such that $r(s,t) > d_{out}(t) \cdot r_{max}$ do
14     $\hat{\pi}(s,t) \leftarrow \hat{\pi}(s,t) + \alpha \cdot r(s,t)$;
15     if $d_{out}(t) \neq 0$ then
16         for each $u \in N_{out}(t)$ do
17             $r(s,u) \leftarrow r(s,u) + (1-\alpha) \cdot \frac{r(s,t)}{d_{out}(t)}$;
18     else
19         $r(s,s) \leftarrow r(s,s) + (1-\alpha) \cdot r(s,t)$;
20     $r(s,t) \leftarrow 0$;
```

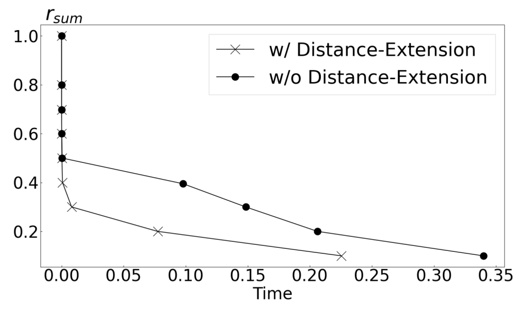


FIGURE 4. $\mathbf{r_{sum}}$ v.s elapsed time (sec) on Pokec dataset.

increases the k value (Lines 3-12). Since vertices in V_k with small k are assigned close vertex IDs and pushing operation on these vertices have large benefits, DEFP can rapidly reduce r_{sum}. Fig. 4 shows the r_{sum} versus the elapsed time. Compared with the result without *Distance-Extension* strategy, we can see that DEFP shows the superiority. DEFP switches to the original FP algorithm shown in Algorithm 1 after k reaches D_s (Lines 13-20).

C. VERTEX-CENTRIC RANDOM WALK

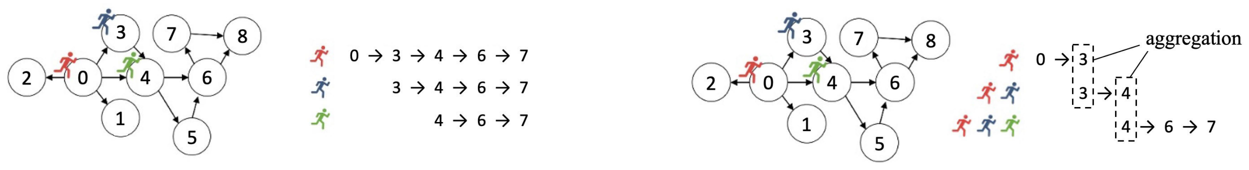
In the MC phase, a vertex v with $r(s,v) > 0$ performs $\lceil r(s,v) \cdot \omega \rceil$ random walks and increases $\hat{\pi}(s,t)$ of the termination vertex t. Letting $I_v = \frac{r(s,v)}{\lceil r(s,v) \cdot \omega \rceil}$, existing methods increase $\hat{\pi}(s,t)$ by I_v. Since I_v depends on $r(s,v)$ and therefore differs for each vertex v, existing methods need to perform all random walks sequentially to memorize the starting vertex of each random walk. In this case, the total number of operations to obtain neighbor vertices is excessively large because each random walk needs to obtain neighbor vertices for every single step. We show an example in Fig. 5. Assume that there are three random walks having similar paths, where $0 \to 3 \to 4 \to 6 \to 7$ (red), $3 \to 4 \to 6 \to 7$ (blue), $4 \to 6 \to 7$ (green). Even though all these random walks pass through vertex 4, 6, and 7 in the same order, we need to obtain neighbor vertices of vertex 4, 6, and 7 three times, respectively.

VCRW is an effective solution for this problem. Our idea is to aggregate random walks at each vertex v so that these aggregated random walks can move by obtaining $N_{out}(v)$ only once. We show an example of the aggregation in Fig. 5(b). The three random walks shown in Fig. 5(a) can reach the termination vertex 7 at the same time. Since it suffices to obtain neighbor vertices at vertex 4, 6, and 7 only once for each, we can reduce the total number of operations to obtain neighbor vertices from twelve in Fig. 5(a) to five in Fig. 5(b).

To realize the aggregation, we need to perform random walks that depend on $r(s,v)$ as little as possible. Therefore, we unify the increment value of $\hat{\pi}(s,t)$ while maintaining accuracy guarantees. Specifically, we unify the increment value of $\lfloor r(s,v) \cdot \omega \rfloor$ random walks out of all $\lceil r(s,v) \cdot \omega \rceil$ random walks. By unifying most of the increment value, we can aggregate a large part of random walks, which significantly reduces the total number of operations to obtain neighbor vertices. Although we still need to perform sequential random walks whose increment value depends on $r(s,v)$, it is guaranteed that each vertex v performs $\lceil r(s,v) \cdot \omega \rceil - \lfloor r(s,v) \cdot \omega \rfloor$ sequential random walks. Since the value of $\lceil r(s,v) \cdot \omega \rceil - \lfloor r(s,v) \cdot \omega \rfloor$ is either 0 or 1, the running cost of sequential random walks is fairly low.

Algorithm 5 shows the pseudo-code of VCRW. VCRW consists of two types of random walk, namely *Independent Random Walk (IRW)* (Lines 2-13), and *Dependent Random Walk (DRW)* (Lines 14-18). In IRW, each vertex v initially has $\lfloor r(s,v) \cdot \omega \rfloor$ random walks (Line 2). A vertex v that has one or more random walks first obtains its own neighbor vertices (Line 4) and decides the next destination of each random walk (Lines 7-8). The destination vertex u chosen by v aggregates a random walk (Line 9). If a random walk terminates at v, v's reserve is increased by $\frac{1}{\omega}$, which is a unified constant value (Lines 10-12). Since we do not need to memorize where each random walk is from, all random walks at v can move by obtaining $N_{out}(v)$ only once. In DRW, each vertex v with $r(s,v) > 0$ performs a sequential random walk only once. Letting $c_v = r(s,v) \cdot \omega - \lfloor r(s,v) \cdot \omega \rfloor$, this sequential random walk increases $\hat{\pi}(s,t)$ of a termination vertex t by $\frac{c_v}{\omega}$ (Lines 14-18).

We show that VCRW returns $\hat{\pi}(s,t)$ satisfying Definition 1 for all $t \in V$. We first introduce a generalization of the Chernoff inequalities (Lemma 1), and subsequently guarantee the accuracy (Theorem 2).



(a) Sequential Random Walk (b) Vertex-Centric Random Walk (VCRW)

FIGURE 5. Assuming that there are three random walks from vertex 0, 3, and 4 that have the same path $4 \longrightarrow 6 \longrightarrow 7$, sequential random walks need to obtain neighbor vertices every time as shown in Fig. 5(a). On the other hand, VCRW can reduce the total number of operations to obtain neighbor vertices by aggregating three random walks at vertex 3 and vertex 4 as shown in Fig. 5(b).

Algorithm 5: Vertex-Centric Random Walk

```
   Input: Graph $G^*$, source vertex $s$, termination
          probability $\alpha$, threshold $r_{max}$
   Output: PPR score $\hat{\pi}(s,t)$ for all $t \in V$
1  $\omega \leftarrow \frac{(2\cdot\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\cdot\mu}$;
   # Independent Random Walk
2  IRW($v$) $\leftarrow \lfloor r(s,v)\cdot\omega \rfloor$ for all $v \in V$;
3  while $\exists v \in V$ such that $IRW(v) > 0$ do
4      $Neighbors \leftarrow N_{out}(v)$;
5      for $i = 1$ to $IRW(v)$ do
6          Generate a random number $r$ between $(0,1)$;
7          if $r > \alpha$ then
8              Choose $u$ randomly from $Neighbors$;
9              IRW($u$) $\leftarrow$ IRW($u$) + 1;
10         else
11             $t \leftarrow v$;
12             $\hat{\pi}(s,t) \leftarrow \hat{\pi}(s,t) + \frac{1}{\omega}$;
13     IRW($t$) $\leftarrow 0$;
   # Dependent Random Walk
14 for each $v \in V$ with $r(s,v) > 0$ do
15     $c_v \leftarrow r(s,v)\cdot\omega - \lfloor r(s,v)\cdot\omega \rfloor$;
16     Generate a sequential random walk from $v$;
17     Let $t$ be the termination vertex;
18     $\hat{\pi}(s,t) \leftarrow \hat{\pi}(s,t) + \frac{c_v}{\omega}$;
```

Lemma 1 (Chernoff Bound [44]): Let $X_1, \ldots, X_n$ be independent random variables with

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

Letting $X = \sum_{i=1}^{w} a_i X_i$ with $a_i > 0$, $\nu = \sum_{i=1}^{w} a_i^2 p_i$, $a = \max\{a_1, \ldots, a_n\}$ and $\lambda \geq 0$, we have

$$Pr[|X - \mathbb{E}[X]| \geq \lambda] \leq 2\exp\left(\frac{-\lambda^2}{2\nu + 2a\lambda/3}\right). \tag{4}$$

Theorem 2: For any vertex t with $\pi(s,t) > \delta$, if $\omega \geq \frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\cdot\delta}$, VCRW returns $\hat{\pi}(s,t)$ satisfying (1) with at least $1 - p_f$ probability.

Proof: Let ω_{sum} be the total number of random walks, and $X_j(t)$ be a random variable as follows:

$$X_j(t) = \begin{cases} 1 & \text{if } j\text{-th random walk terminates at } t, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

To apply Lemma 1, let $X = \sum_{j=1}^{\omega_{sum}} a_j \cdot X_j(t)$, and $\nu = \sum_{j=1}^{\omega_{sum}} a_j^2 \cdot \mathbb{E}\left[X_j(t)\right]$, where a_j takes value 1 if j-th random walk starting from v is IRW, and value $r(s,v)\cdot\omega - \lfloor r(s,v)\cdot\omega \rfloor$ if j-th random walk starting from v is DRW.

By definition, $a_j \leq 1$. Therefore, $\nu \leq \mathbb{E}[X]$ and $a \leq 1$, where $a = \max\{a_1, \ldots, a_{\omega_{sum}}\}$. Applying them to Lemma 1, we have

$$Pr[|X - \mathbb{E}[X]| \geq \lambda] \leq 2\exp\left(\frac{-\lambda^2}{2\mathbb{E}[X] + 2\lambda/3}\right). \tag{6}$$

Let $c_v = r(s,v)\cdot\omega - \lfloor r(s,v)\cdot\omega \rfloor$, and $Y_k(t)$ denote a random variable that takes value 1 if k-th random walk starting from v terminates at t, and value 0 otherwise. By definition,

$$\mathbb{E}[Y_k(t)] = \pi(v,t). \tag{7}$$

Then we have,

$$\begin{aligned} \mathbb{E}\left[\frac{X}{\omega}\right] &= \mathbb{E}\left[\sum_{j=1}^{\omega_{sum}} \frac{a_j}{\omega} \cdot X_j(t)\right] \\ &= \mathbb{E}\left[\sum_{v\in V}\left(\sum_{k=1}^{\lfloor r(s,v)\cdot\omega\rfloor} \frac{1}{\omega}\cdot Y_k(t) + \frac{c_v}{\omega}\cdot Y_k(t)\right)\right] \\ &= \mathbb{E}\left[\sum_{v\in V} r(s,v)\cdot Y_k(t)\right] \\ &= \sum_{v\in V} r(s,v)\cdot\pi(v,t). \end{aligned} \tag{8}$$

Observing that $\sum_{j=1}^{\lfloor r(s,v)\cdot\omega\rfloor} \frac{1}{\omega}\cdot Y_k(t)$ is exactly the amount of increment that $\hat{\pi}(s,t)$ receives from a vertex v during IRW, and $\frac{c_v}{\omega}\cdot Y_k(t)$ is that of during DRW. Hence, we can rewrite (2) and (3) as follows.

$$\pi(s,t) = \pi(s,t)^\circ + \frac{1}{\omega}\cdot\mathbb{E}[X] \tag{9}$$

$$\hat{\pi}(s,t) = \pi(s,t)^\circ + \frac{1}{\omega}\cdot X \tag{10}$$

Based on (9) and (10), we have $\frac{1}{\omega} \cdot (\mathbb{E}[X] - X) = \pi(s,t) - \hat{\pi}(s,t)$, and $\mathbb{E}[X] \leq \omega \cdot \pi(s,t)$. Therefore, (6) can be rewritten as

$$Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \lambda/\omega] \leq 2\exp\left(\frac{-\lambda^2}{2 \cdot \omega \cdot \pi(s,t) + 2\lambda/3}\right). \tag{11}$$

Finally, letting $\lambda = \omega \cdot \epsilon \cdot \pi(s,t)$, $\omega \geq \frac{(2\cdot\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\cdot\delta}$, and $\pi(s,t) > \delta$, (11) follows

$$Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon \cdot \pi(s,t)] < p_f. \tag{12}$$

This completes the proof. □

D. RUNNING TIME ANALYSIS

It has been proved that the expected running time of the FP algorithm shown in Algorithm 1 can be bounded by $O\left(m \cdot \log\frac{1}{m \cdot r_{max}}\right)$ [15]. Since the proposed method follows Algorithm 1, we can also define the expected running time of DEFP as $O\left(m \cdot \log\frac{1}{m \cdot r_{max}}\right)$.

Letting 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 the MC phase is determined by the total number of operations to obtain neighbor vertices. Therefore, we estimate this number to define the expected running time of VCRW.

We first investigate the expected running time of DRW. In DRW, each vertex 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 investigate 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 vertices at the beginning of the k-th iteration, where a vertex v is active if v has one or more random walks. Therefore, $I^{(k)}$ consists of vertices that still have random walks after all vertices in $I^{(k-1)}$ move their random walks. Initially, $I^{(0)}$ contains every vertex v with $\lfloor r(s,v)\cdot\omega \rfloor \geq 1$. In addition, we denote $\omega_{sum}^{(k)}$ as the remaining number of random walks at the beginning of the k-th iteration. Based on these definitions, we prove the following theorem.

Theorem 3: The expected running time of IRW is $O(n \cdot \log(m \cdot r_{max} \cdot \omega))$.

Proof:

Considering the $(k+1)$-th iteration, we have

$$\omega_{sum}^{(k+1)} \leq (1-\alpha) \cdot \omega_{sum}^{(k)}. \tag{13}$$

Based on this, we can easily get

$$\omega_{sum}^{(k)} \leq (1-\alpha)^k \cdot \omega_{sum}^{(0)}. \tag{14}$$

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

$$\omega_{sum}^{(0)} = \sum_{v \in V} \lfloor r(s,v) \cdot \omega \rfloor \leq \sum_{v \in V} \lfloor r_{max} \cdot d_{out}(v) \cdot \omega \rfloor \leq m \cdot r_{max} \cdot \omega. \tag{15}$$

We assume that IRW finishes when $\omega_{sum}^{(k)} < 1$. By combining (14) and (15), we know that IRW needs $K = O(\log(m \cdot r_{max} \cdot \omega))$ iterations to satisfy the condition $\omega_{sum}^{(k)} < 1$.

In the k-th iteration, $\left|I^{(k)}\right|$ vertices are still active. Observing that $\left|I^{(k)}\right| \leq n$, the total number of operations to obtain neighbor vertices follows

$$\sum_{k=0}^{K-1} \left|I^{(k)}\right| \leq \sum_{k=0}^{K-1} n = n \cdot K. \tag{16}$$

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

V. EVALUATION

A. EXPERIMENTAL SETUP

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

Name	n	m	m/n	Type
WebStan	282 K	2.31 M	8.20	Directed
DBLP	317 K	2.10 M	6.62	Undirected
Pokec	1.63 M	30.6 M	18.8	Directed
LiveJournal	4.85 M	68.4 M	14.1	Directed
Orkut	3.07 M	234 M	76.3	Undirected
Twitter	41.7 M	1.46 B	35.3	Directed

We conducted all experiments on a Linux 20.04 server with dual Intel Xeon E5-2643 processors and 94GiB memory. The size of the L1 data cache, L1 instruction cache, L2 cache, and L3 cache were 384KiB, 384KiB, 3MiB, and 50MiB, respectively. All algorithms were implemented with C++ and compiled with G++ 9.4.0 using the -O3 optimization.

1) REAL-WORLD GRAPHS

We used six real-world graphs; web pages from Stanford University (WebStan) [45], DBLP collaboration network (DBLP) [46], Pokec social network (Pokec) [43], LiveJournal social network (LiveJournal) [47], Orkut social network (Orkut) [46], and Twitter social network (Twitter) [48]. All the graphs are typically used as the benchmarks and available on the Stanford SNAP library.[2] Details of all the graphs are shown in Table 2.

2) EXPERIMENTS

We compared the proposed method with three index-free methods: FORA [16], ResAcc [14], and SpeedPPR [15], and the state-of-the-art index-oriented method FORA+ [23]. The computational efficiency was measured through the overall

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

running time and the cache performance. Before starting the experiments, we shuffled the vertex IDs randomly. This is because the given vertex IDs have been modified by publishers, which might influence the experimental results unintentionally [19]. After shuffling the vertex IDs, we generated 50 query source vertices uniformly at random. We report the average results of these 50 vertices. To generate random numbers efficiently, we used the latest version of SIMD-oriented Fast Mersenne Twister Library.[3]

As for cache performance analytics, we used the perf tool.[4] We measured five events on processors: L1-r, L1-m, L3-r, L3-m, and Total-m. L1-r denotes the number of L1 cache references and L1-m denotes the number of L1 cache misses. L1-r is equal to the total number of cache references because processors first reference the L1 cache. L1-r and L1-m can be measured with the perf options `L1-dcache-loads` and `L1-dcache-misses`, respectively. Similarly, L3-r and L3-m denote the corresponding numbers at the L3 cache, and these numbers can be measured with `LLC-loads` and `LLC-misses` options, respectively. Total-m denotes the percentage of the total cache misses to the total cache references, such that Total-m = L3-m/L1-r. Note that we made sure that all caches were initialized before each experiment by executing `sysctl -w vm.drop_caches=3`.

3) PARAMETER SETTINGS

By default, we set $\alpha = 0.2$, $\delta = \frac{1}{n}$, $p_f = \frac{1}{n}$, and $\epsilon = 0.5$ following the existing methods. Specifically, we set $r_{max} = \frac{1}{\omega}$, where $\omega = \frac{(2\cdot\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\cdot\delta}$, as default because this setting can theoretically establish the lowest expected running time [15]. Moreover, ResAcc has two additional parameters h and r_{max}^{hop}. In a nutshell, ResAcc first performs the FP algorithm shown in Algorithm 1 with $r_{max} = r_{max}^{hop}$ on vertices within h distances from the source vertex s, and accumulates residues at s. After distributing accumulated residues, ResAcc continues running Algorithm 1 with the entire graph. We empirically set h and r_{max}^{hop} to realize the best performance.

B. OVERALL RUNNING TIME

Fig. 6 shows the overall running time of the index-free methods in log scale. The running time of the proposed method was smaller than the existing methods on all the datasets except for WebStan. On WebStan, the proposed method slightly underperformed the existing methods. However, this level of performance gap is negligible because the overall running time on WebStan is fairly low. Compared with FORA, ResAcc, and SpeedPPR, the average speedup of the proposed method was 3.2×, 3.9×, and 2.5×, and the maximum speedup was 4.5×, 4.7×, and 3.9×, respectively. It is worth pointing out that the proposed method was more effective on larger graphs. The proposed method was over ten seconds faster on LiveJournal, over 50 seconds faster on Orkut, and over 400 seconds faster on Twitter than the fastest existing method SpeedPPR. This shows a considerable superiority of the proposed method in terms of the computational efficiency.

To analyze the overhead of the reordering procedure, we further examined the breakdown of the overall running time. The overall running time is divided into reordering time and PPR computation time. Fig. 7 shows the corresponding results. On all the datasets, reordering procedure completed faster than PPR computation. Moreover, we defined the overhead of reordering procedure as the ratio of reordering time to PPR computation time. Except for WebStan, this ratio was 0.24 on average. This low overhead enhances the computational efficiency of the proposed method.

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

[4]https://perf.wiki.kernel.org/index.php

TABLE 3. Cache performance on DBLP ($K = 10^3$, $M = 10^6$).

Name	L1-r	L1-m	L3-r	L3-m	Total-m
FORA	224 M	54 M	36 M	529 K	**0.24%**
ResAcc	228 M	54 M	35 M	884 K	0.39%
SpeedPPR	**150 M**	34 M	25 M	**522 K**	0.35%
Proposed	222 M	**21 M**	**10 M**	625 K	0.28%

TABLE 4. Cache performance on LiveJournal ($M = 10^6$, $B = 10^9$).

Name	L1-r	L1-m	L3-r	L3-m	Total-m
FORA	8.1 B	3.2 B	2.4 B	1.1 B	13.6%
ResAcc	8.0 B	3.2 B	2.4 B	1.1 B	13.8%
SpeedPPR	**4.3 B**	1.7 B	1.2 B	530 M	12.3%
Proposed	5.9 B	**1.2 B**	**768 M**	**194 M**	**3.3%**

TABLE 5. Cache performance on Twitter ($B = 10^9$).

Name	L1-r	L1-m	L3-r	L3-m	Total-m
FORA	97 B	40 B	34 B	16 B	16.8%
ResAcc	113 B	45 B	39 B	18 B	15.9%
SpeedPPR	**71 B**	29 B	24 B	12 B	16.6%
Proposed	81 B	**17 B**	**13 B**	**4 B**	**5.2%**

C. CACHE PERFORMANCE

Tables 3, 4, and 5 show the cache performance on DBLP (small size graph), LiveJournal (medium size graph), and Twitter (large size graph), respectively. The best result in each column is highlighted in **bold**. The proposed method greatly reduced both L1-m and L3-m on LiveJournal and Twitter. Notably, Total-m on Twitter was up to 11.6% lower than the existing methods. Since the cache reference is over two orders of magnitude faster than the main memory reference, the cache miss reduction primarily accelerates the proposed method. These results on LiveJournal and Twitter are consistent with the overall running time results shown in Fig. 6. While the proposed method succeeded in reducing L1-m on DBLP, L3-m was nearly identical to the existing methods. This is mainly because the graph size of DBLP is so small that the L3 cache can hold a large part of data. Interestingly, the proposed method reduced L1-r compared to FORA and ResAcc, even though the proposed method needs to go through the reordering procedure before the FP phase.

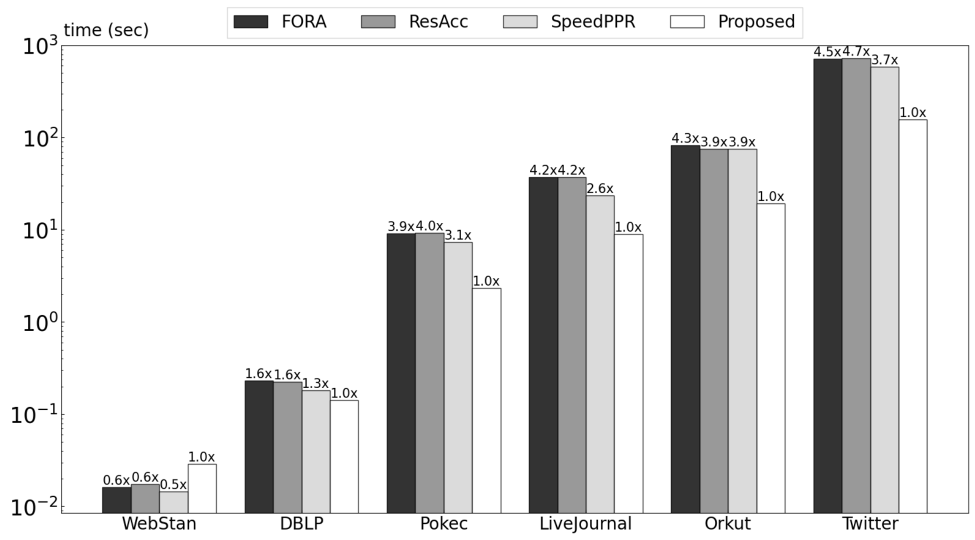


FIGURE 6. The overall running time on each dataset. The number d.dx over each bar means that Proposed is d.dx faster than the others.

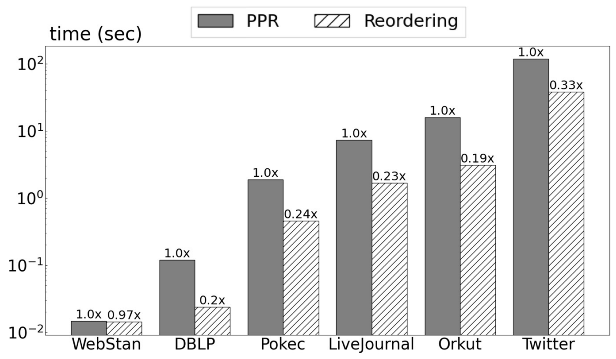


FIGURE 7. The PPR computation time and the reordering time of the proposed method.

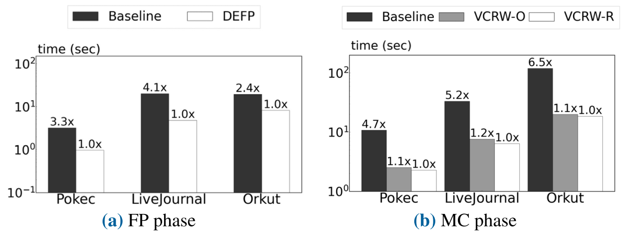


FIGURE 8. Effect of each optimization on the running time.

This shows that DEFP and VCRW improve the computational efficiency.

D. EFFECT OF EACH OPTIMIZATION

We investigated the respective effects of DEFP and VCRW on the running time. Fig. 8 shows the running time of each approach in log scale on Pokec, LiveJournal, and Orkut. We first compare the running time of Algorithm 1 (Baseline) and DEFP in Fig. 8(a). We measured the running time until there are no active vertices with respect to the default r_{max}. Compared with Baseline, DEFP was 3.3×, 4.1×, and 2.4× faster on Pokec, LiveJournal, and Orkut, respectively. Considering that the running time of DEFP includes the reordering time, this speedup shows that DEFP can proceed the FP phase efficiently.

Next, we compare the running time in the MC phase. Since the proposed method performs VCRW after conducting the reordering in the FP phase, the reordered data layout might have a positive effect on the running time in addition to the aggregation technique. To clearly separate the effect of the aggregation technique and the reordering, we measured the running time of VCRW on original layout (VCRW-O) and on reordered layout (VCRW-R). Fig. 8(b) shows the running time of Baseline, VCRW-O, and VCRW-R, respectively. Note that Baseline represents the original approach that performs random walks sequentially. The running time was measured up to the point where each vertex v has performed $\lceil r_{max} \cdot d_{out}(v) \cdot \omega \rceil$ random walks. We can see that VCRW-R was 4.7×, 5.2×, and 6.5× faster than Baseline on Pokec, LiveJournal, and Orkut, respectively, which shows a significant speedup. On the other hand, VCRW-R was slightly faster than VCRW-O on the three datasets. Therefore, we confirm that the speedup of the MC phase by VCRW is mostly due to the aggregation technique.

E. EFFECT OF SETTING r_{max}

We examined the effect of setting r_{max} on the overall running time using Pokec and Twitter. Recall that r_{max} determines when the FP phase switches to the MC phase. Theoretically, the default setting can achieve the fastest running time. However, previous works have observed that the best setting differs from the default setting and the running time is too sensitive to r_{max}. Therefore, we measured the overall running

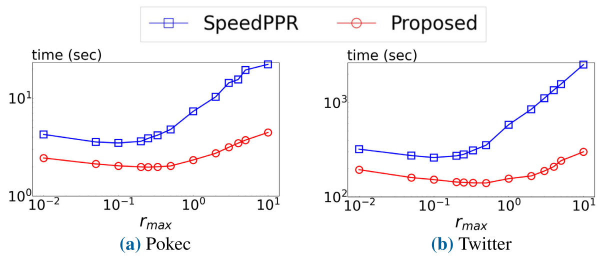


FIGURE 9. The overall running time with different r_{max} $(\times r^{\circ}_{max})$.

time against different r_{max} varying from $10^{-2} \times r^{\circ}_{max}$ to $10^{1} \times r^{\circ}_{max}$, where r°_{max} denotes the default setting.

Fig. 9 shows the corresponding results of SpeedPPR and the proposed method. As previous works have reported, we observed that r°_{max} cannot achieve the fastest running time on both Pokec and Twitter. The proposed method (resp. SpeedPPR) was fastest with $r_{max} = 0.25 \times r^{\circ}_{max}$ and $r_{max} = 0.5 \times r^{\circ}_{max}$ (resp. $r_{max} = 0.1 \times r^{\circ}_{max}$ and $r_{max} = 0.1 \times r^{\circ}_{max}$) on Pokec and Twitter, respectively. At the fastest point, SpeedPPR was 2.1× and 2.2× faster than the default setting on Pokec and Twitter, respectively, which shows SpeedPPR is sensitive to r_{max}. On the other hand, in the proposed method, the difference between the fastest running time and the running time with r°_{max} was 1.2× and 1.1× on Pokec and Twitter, respectively. These results show that the proposed method is less sensitive to r_{max} than SpeedPPR.

F. EFFECT OF PARAMETER SETTINGS FOR ACCURACY GUARANTEES

Recall that the parameters ϵ and δ determine the condition of Approximate SSPPR query. Specifically, the value of ϵ determines the acceptable relative error and the value of δ determines the applicable scope of accuracy guarantees. According to numerous demands from applications, PPR queries are conducted with various parameters settings. Motivated by this, we evaluated the overall running time against different ϵ and δ.

Fig. 10 shows the overall running time against different ϵ varying from 0.5 to 0.1. We can see that the proposed method significantly outperformed the existing methods with any ϵ on all the datasets except for WebStan. Note that both the proposed method and SpeedPPR showed a linear increase on the overall running time. On the other hand, the overall running time of FORA and ResAcc sharply increased with a decrease of ϵ, especially on Twitter. This is mainly due to the difference in the way to access active vertices during the FP phase. Since FORA and ResAcc adopt a queue-based implementation, active vertices are accessed in a nonconsecutive manner with respect to vertex IDs, which leads to irregular memory access. Alternatively, the proposed method and SpeedPPR adopt an array-based implementation, and both methods can therefore access the active vertices consecutively in the order of vertex IDs.

Fig. 11 shows the overall running time against different δ. Letting $\delta^{\circ} = \frac{1}{n}$ be the default parameter, we measured the overall running time with $\delta = \frac{\delta^{\circ}}{\delta^{*}}$. Note that we varied δ^{*} from $\{1, 5, 10\}$. In addition, Table 6 shows the percentage of vertices that are the target of accuracy guarantees. To calculate the percentage, we need to know the ground truth score $\pi(s, t)$ for all $t \in V$. Therefore, we ran Algorithm 1 until r_{sum} satisfied $r_{sum} < \theta$, where $\theta = \min\{10^{-8}, \frac{1}{m}\}$, and we then regarded the returned result as the ground truth. Similar to the results varying the parameter ϵ as shown in Fig. 10, the proposed method outperformed the existing methods. Notably, the proposed method with $\delta^{*} = 10$ was faster than the existing methods with $\delta^{*} = 1$ on larger graphs: LiveJournal, Orkut, and Twitter. Therefore, the proposed method can obtain more accurate results than existing methods with the same running cost.

TABLE 6. The percentage of vertices that are the target of accuracy guarantees.

Name	$\delta^{*} = 1$	$\delta^{*} = 5$	$\delta^{*} = 10$
WebStan	0.43%	0.93%	1.2%
DBLP	2.1%	8.3%	14%
Pokec	6.0%	26%	38%
LiveJournal	8.9%	33%	46%
Orkut	11%	58%	76%
Twitter	4.5%	12%	17%

G. COMPARISON WITH INDEX-ORIENTED METHOD

Fig. 12 shows the overall running time of the proposed method and FORA+ [23] on LiveJournal, Orkut, and Twitter against different ϵ varying from 0.5 to 0.1. We obtained the source code of FORA+ from the authors.[5] For FORA+, we generated the index with smallest $\epsilon = 0.1$ because the index generated with $\epsilon = \epsilon_{High}$ cannot be reused to answer PPR queries with a smaller $\epsilon = \epsilon_{Low} < \epsilon_{High}$.

Overall, FORA+ was faster than the proposed method because FORA+ simply reads the index in the MC phase. Specifically, FORA+ outperformed the proposed method with larger ϵ, and FORA+ was 2.9×, 3.4×, and 1.7× faster with $\epsilon = 0.5$ on LiveJournal, Orkut, and Twitter, respectively. One interesting finding is that the proposed method achieved a comparable running time with smaller ϵ. In particular, the proposed method outperformed FORA+ on LiveJournal and Twitter under rigorous accuracy guarantees, i.e., $\epsilon = 0.1$. It is worth pointing out that the proposed method is over 100 seconds faster than FORA+ with $\epsilon = 0.1$ on Twitter.

Table 7 summarizes the online running time with $\epsilon = 0.1$, offline index generation time, the index size, and the graph size. While FORA+ requires $O\left(\frac{n \cdot \log n}{\epsilon}\right)$ space consumption and impractical index generation time, our index-free method has zero precomputation costs. We can see that FORA+ consumed minutes or hours of precomputation time and huge space overheads beyond the graph size, which is unacceptable in real-world scenarios. Assuming that users request rigorous accuracy guarantees and set a small ϵ, the performance of FORA+ is apparently degraded in terms of online query time

[5]https://github.com/wangsibovictor/fora

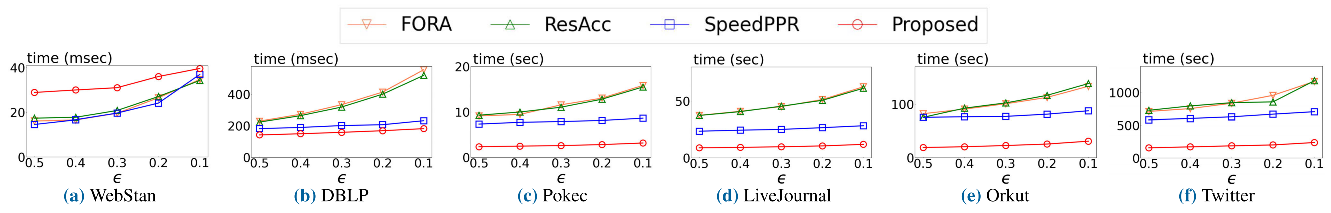


FIGURE 10. The overall running time against different ϵ.

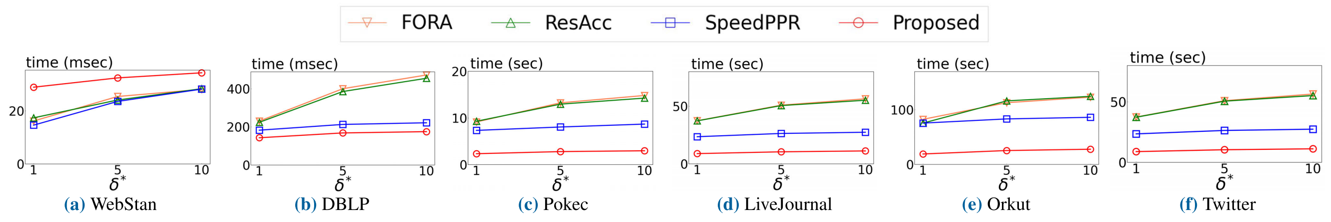


FIGURE 11. The overall running time with $\delta = \frac{\delta^\circ}{\delta^*}$, where δ° is the default parameter.

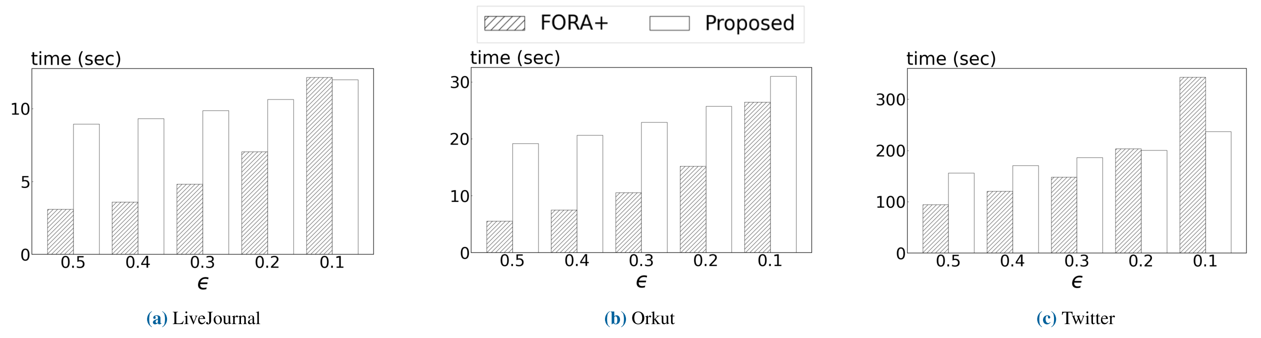


FIGURE 12. The overall running time of FORA+ and the proposed method against different ϵ.

TABLE 7. Performance comparison with FORA+ and the proposed method. The index of FORA+ was generated with $\epsilon = 0.1$.

Name	Running time (sec)		Index generation time (sec)		Index size (Byte)		Graph size (Byte)
	FORA+	Proposed	FORA+	Proposed	FORA+	Proposed	
LiveJournal	12.1	**11.9**	465	**0**	3.3 G	**0**	1.1 G
Orkut	**26.5**	31.0	1032	**0**	4.8 G	**0**	1.7 G
Twitter	343	**237**	14107	**0**	48 G	**0**	25 G

and offline precomputation overheads. These results verify that the proposed method can achieve significant computational efficiency without any precomputations.

H. COMPARISON WITH GORDER

Finally, we compared the proposed method with SpeedPPR incorporated with the state-of-the-art graph reordering method Gorder [18]. Gorder is the representative heavyweight reordering method that achieves remarkable speedup by deeply inspecting the connectivity of vertices. Fig. 13 shows the overall running time of SpeedPPR, SpeedPPR+Gorder, and the proposed method on LiveJournal. The result of SpeedPPR+Gorder represents the running time of SpeedPPR on the reordered LiveJournal dataset. We observe that SpeedPPR+Gorder

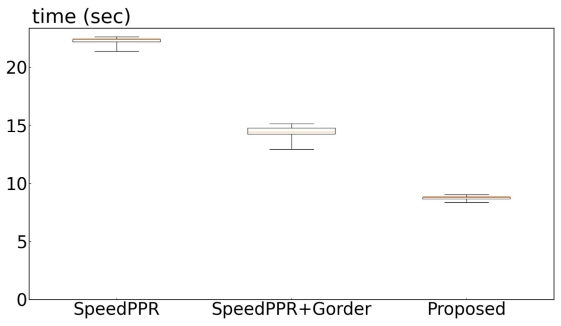


FIGURE 13. The boxplot of the overall running time on LiveJournal.

greatly outperformed SpeedPPR, and the average running time of SpeedPPR+Gorder was 1.5× smaller than

SpeedPPR. Notably, the proposed method was still faster than SpeedPPR+Gorder. Note that the result of the proposed method includes reordering time. On the other hand, the result of SpeedPPR+Gorder excludes it. Considering that Gorder took 70.1 seconds to complete the reordering procedure on LiveJournal, we confirm that the proposed method is more practical than SpeedPPR+Gorder. Interestingly, we find that the running time of the proposed method is stable. The standard deviations of three methods were 0.29, 0.46, and 0.16, respectively, which shows that the proposed method performed stably regardless of the characteristics of source vertices. This is mainly because the proposed method always reorders the data layout according to a given source vertex.

VI. CONCLUSION

We proposed a cache-efficient approach for fast index-free PPR computation. The proposed method significantly accelerates the FORA algorithm by optimizing the computational procedure of both FP and MC phases. Experiment results using real-world graphs showed that the proposed method reduces the cache miss ratio by up to 11.6% over existing methods on the largest dataset. As a result, the proposed method was faster than the fastest existing method by an average of 2.5× and a maximum of 3.9×. Moreover, the proposed method outperformed the state-of-the-art index-oriented method in query time under rigorous accuracy guarantees. Since various real-world applications, including link prediction, community detection, and social recommendation, utilize the PPR scores, the proposed method can be used to further improve the efficiency of these applications.

REFERENCES

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford InfoLab, Stanford, CA, USA, Tech. Rep. 1999-66, 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. 4th Int. Workshop Adversarial Inf. Retr. Web*, Apr. 2008, pp. 69–76.

[3] L. Backstrom and J. Leskovec, "Supervised random walks: Predicting and recommending links in social networks," in *Proc. 4th ACM Int. Conf. Web Search Data Mining*, Feb. 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. 22nd Int. Conf. World Wide Web*, May 2013, pp. 505–514.

[5] Y. Gao, X. Yu, and H. Zhang, "Overlapping community detection by constrained personalized PageRank," *Exp. Syst. Appl.*, vol. 173, Jul. 2021, Art. no. 114682.

[6] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2017, pp. 555–564.

[7] J. J. Whang, D. F. Gleich, and I. S. Dhillon, "Overlapping community detection using neighborhood-inflated seed expansion," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 5, pp. 1272–1284, May 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. Int. Conf. Mach. Learn. (ICML)*, vol. 2018, pp. 5453–5462.

[9] J. Klicpera, A. Bojchevski, and S. Günnemann, "Predict then propagate: Graph neural networks meet personalized pagerank," in *Proc. 8th Int. Conf. Learn. Represent.*, 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. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 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 Int. Conf. Manage. Data*, May 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 34th Int. Conf. Data Eng. (ICDE)*, Apr. 2018, pp. 1132–1143.

[14] 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 36th Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 913–924.

[15] H. Wu, J. Gan, Z. Wei, and R. Zhang, "Unifying the global and local approaches: An efficient power iteration with forward push," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 1996–2008.

[16] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang, "FORA: Simple and effective approximate single-source personalized PageRank," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2017, pp. 505–514.

[17] Y. Lim, U. Kang, and C. Faloutsos, "SlashBurn: Graph compression and mining beyond caveman communities," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 12, pp. 3077–3089, Dec. 2014.

[18] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 1813–1828.

[19] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 22–31.

[20] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 293–302.

[21] V. Balaji and B. Lucia, "When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Sep. 2018, pp. 203–214.

[22] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2019, pp. 1–13.

[23] 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 Trans. Database Syst.*, vol. 44, no. 4, pp. 1–37, Dec. 2019.

[24] K. Tsuchida, N. Matsumoto, and K. Kaneko, "Node-centric random walk for fast index-free personalized PageRank," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, 2023.

[25] J. Siek, L. Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Reading, MA, USA: Addison-Wesley, 2002.

[26] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using PageRank vectors," in *Proc. 47th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*, Oct. 2006, pp. 475–486.

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

[28] G. Jeh and J. Widom, "Scaling personalized web search," in *Proc. 12th Int. Conf. World Wide Web (WWW)*, 2003, pp. 271–279.

[29] K. Shin, J. Jung, S. Lee, and U. Kang, "BEAR: Block elimination approach for random walk with restart on large graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 1571–1585.

[30] R. Andersen, C. Borgs, J. Chayes, J. Hopcroft, V. Mirrokni, and S.-H. Teng, "Local computation of PageRank contributions," *Internet Math.*, vol. 5, nos. 1–2, pp. 23–45, Jan. 2008.

[31] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J.-R. Wen, "TopPPR: Top-k personalized PageRank queries with precision guarantees on large graphs," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2018, pp. 441–456.

[32] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka, "Efficient personalized PageRank with accuracy assurance," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2012, pp. 15–23.

[33] S. Luo, X. Xiao, W. Lin, and B. Kao, "BATON: Batch one-hop personalized PageRanks with efficiency and accuracy," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 10, pp. 1897–1908, Oct. 2020.

[34] M. Liao, R.-H. Li, Q. Dai, and G. Wang, "Efficient personalized PageRank computation: A spanning forests sampling based approach," in *Proc. Int. Conf. Manage. Data*, Jun. 2022, pp. 2048–2061.

[35] S. Chakrabarti, "Dynamic personalized pagerank in entity-relation graphs," in *Proc. 16th Int. Conf. World Wide Web*, May 2007, pp. 571–580.

[36] M. Yoon, W. Jin, and U. Kang, "Fast and accurate random walk with restart on dynamic graphs with guarantees," in *Proc. World Wide Web Conf. World Wide Web (WWW)*, 2018, pp. 409–418.

[37] D. Mo and S. Luo, "Agenda: Robust personalized PageRanks in evolving graphs," in *Proc. 30th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2021, pp. 1315–1324.

[38] H. Wang, Z. Wei, J. Gan, Y. Yuan, X. Du, and J.-R. Wen, "Edge-based local push for personalized PageRank," *Proc. VLDB Endowment*, vol. 15, no. 7, pp. 1376–1389, Mar. 2022.

[39] B. Bahmani, K. Chakrabarti, and D. Xin, "Fast personalized PageRank on MapReduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2011, pp. 973–984.

[40] T. Guo, X. Cao, G. Cong, J. Lu, and X. Lin, "Distributed algorithms on exact personalized PageRank," in *Proc. ACM Int. Conf. Manage. Data*, May 2017, pp. 479–494.

[41] G. Hou, X. Chen, S. Wang, and Z. Wei, "Massively parallel algorithms for personalized PageRank," *Proc. VLDB Endowment*, vol. 14, no. 9, pp. 1668–1680, May 2021.

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

[43] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *Proc. Int. Sci. Conf. Int. Workshop Present Day Trends Innov.*, 2012, pp. 1–6.

[44] F. Chung and L. Lu, "Concentration inequalities and Martingale inequalities: A survey," *Internet Math.*, vol. 3, no. 1, pp. 79–127, 2006.

[45] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Math.*, vol. 6, no. 1, pp. 29–123, Jan. 2009.

[46] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, Jan. 2015.

[47] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2006, pp. 44–54.

[48] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web*, Apr. 2010, pp. 591–600.

KOHEI TSUCHIDA received the B.E. degree from the Faculty of Science and Technology, Keio University, in 2021, where he is currently pursuing the master's degree with the Graduate School of Science and Technology. His research interest includes graph analysis.

NAOKI MATSUMOTO received the M.S. and Ph.D. degrees from the Graduate School of Environment and Information Sciences, Yokohama National University. He is currently a Project Assistant Professor with the Research Institute for Digital Media and Content, Keio University. In 2019, he joined at the Research Institute for Digital Media and Content (DMC). His research interests include graph theory and combinatorial game theory and their applications.

ANDREW SHIN received the M.S. degree from the Interdisciplinary Graduate School of Science and Engineering, Tokyo Institute of Technology, and the Ph.D. degree from the Graduate School of Information Science and Technology, The University of Tokyo. He is currently a Project Assistant Professor with the Research Institute for Digital Media and Content, Keio University. Prior to joining Keio University, he was a Research Engineer at Sony Group Corporation Research and Development Center. His research interests include computer vision and natural language processing.

KUNITAKE KANEKO (Member, IEEE) received the M.S. degree from the Graduate School of Engineering, The University of Tokyo, and the Ph.D. degree from the Graduate School of Information Science and Technology. He is currently an Associate Professor with the Faculty of Science and Technology, Keio University. In 2006, he joined at the Research Institute for Digital Media and Content (DMC), Keio University. His major research interest is data networking as a new generation network architecture. He is a member of SMPTE, IPSJ, and IEICE.