Index-Free Approach with Theoretical Guarantee for Efficient Random Walk with Restart Query

Dandan Lin[1], Raymond Chi-Wing Wong[1], Min Xie[2], Victor Junqiu Wei[3]
[1]*The Hong Kong University of Science and Technology,* [2]*Shenzhen Institute of Computing Sciences, Shenzhen University,* [3]*Noah's Ark Lab of Huawei*
[1]{dlinaf, raywong}@cse.ust.hk, [2]xiemin@sics.ac.cn, [3]wei.junqiu1@huawei.com

***Abstract*—Due to the prevalence of graph data, graph analysis is very important nowadays. One popular analysis on graph data is Random Walk with Restart (RWR) since it provides a good metric for measuring the proximity of two nodes in a graph. Although RWR is important, it is challenging to design an algorithm for RWR. To the best of our knowledge, there are no existing RWR algorithms which, at the same time, (1) are index-free, (2) return answers with a theoretical guarantee and (3) are efficient. Motivated by this, in this paper, we propose an index-free algorithm called *Residue-Accumulated approach* (ResAcc) which returns answers with a theoretical guarantee efficiently. Our experimental evaluations on large-scale real graphs show that *ResAcc* is up to 4 times faster than the best-known previous algorithm, guaranteeing the same accuracy. Under typical settings, the best-known algorithm ran around 1000 seconds on a large dataset containing 41.7 million nodes, which is too time-consuming, while *ResAcc* finished in 275 seconds with the same accuracy. Moreover, *ResAcc* is up to 6 orders of magnitude more accurate than the best-known algorithm in practice with the same execution time, which is considered as a substantial improvement.**

I. Introduction

The node-to-node proximity captures the relevance between two nodes in a graph and has been recognized as an important research problem in the data mining community [22], [3], [23], [6], [10]. *Random Walk with Restart* (RWR) is a widely adopted proximity measure due to its ability of considering both the local structure and global structure of the graph. Specifically, given a graph G and a pair of nodes, namely s and t in G, the RWR value $\pi(s,t)$ is defined as the probability that a random walk starting from s (the *source node*) terminates at t (the *target node*), which reflects the relevance of t with respect to (w.r.t) s. One useful query of RWR is the *single-source* RWR (SSRWR) query, which takes as input a source node s and returns the RWR values of all nodes in the graph w.r.t s.

The SSRWR query has many real-world applications. One application is for improving the quality of community detection in networks [5], [30], [7], [12]. In addition, SSRWR queries are widely used for real-time recommendation systems [8], [18], [16], [24] (which recommends to a user items that are similar to those the user has liked previously), and *friend-suggestion* on social networks (which recommends to a user some friends who have high relevance to the user).

Although the SSRWR query is widely needed, it is challenging to design an algorithm for effectively computing RWR values quickly. We summarize 3 challenges here. The first challenge is that adopting an index-oriented approach is too costly for SSRWR. That is, the index-oriented approaches require huge time overheads, high memory cost (in the online query phase), and bulky space cost (of the offline indexing structures), leading them infeasible to be applied to dynamic graphs. This challenge motivates us to design an index-free approach in this paper.

The second challenge is that computing exact RWR values is computationally expensive. Among all existing algorithms, *Inverse* [23] is the only one that computes the exact RWR values. Since *Inverse* needs to compute the inverse of an $(n \times n)$ matrix, where n is the number of nodes, it takes $O(n^{2.373})$ time cost, which is unaffordable at all when n is large. This challenge motivates us to design an algorithm returning an approximate solution with a theoretical error bound.

The third challenge is that it is expected to answer the SSRWR query efficiently in many applications like the overlapping community detection mentioned earlier, which is much challenging when we address the above 2 challenges. In the literature, all index-free approaches returning an approximate solution with a theoretical error bound [20], [9], [17], [28], [29] could not answer the SSRWR query efficiently. Among all these approaches, *FORA* [28] has the best performance in the query phase in terms of accuracy and efficiency. Unfortunately, our experimental results show that *FORA* took around 1,000 seconds in Twitter containing only 41.7 million nodes for the SSRWR query. It could not meet the efficient requirement for the real-world applications where the graph, such as Instagram containing 1 billion nodes, is more large-scale than Twitter.

Motivated by the above 3 challenges, in this paper, we design an algorithm called **Residue-Accumulated approach** (*ResAcc*) which satisfies the following requirements.

- **Index-free.** It does not incur any burden to the data management system (i.e., index construction and maintenance).
- **Output-bound.** It outputs the estimated RWR values with accuracy guarantee.
- **High-efficiency.** It is computationally efficient.

However, none of the existing algorithms satisfy all the above 3 requirements simultaneously as shown in Table I.

Our contributions. The following shows our major contributions. (1) Firstly, we propose an index-free algorithm, *ResAcc*, satisfying the 3 requirements simultaneously, by incorporating a novel and highly efficient technique called h-HopFWD where h is a parameter. (2) Secondly, we prove that *ResAcc* can guarantee the user-specified accuracy with $(1-p_f)$ probability

TABLE I
COMPARISON AMONG EXISTING ALGORITHMS FOR THE SSRWR QUERY.

Approach	Technique	Algorithm	Error Bound	Efficiency
Index-oriented	Iterative-based	*TPA* [31]	Additive	Medium
	Matrix-based	*B-LIN* [23]	Not given	Slow
		QR [11]	Not given	Slow
		BEAR [22]	Relative	Medium
		BePI [14]	Relative	Medium
	Monte-Carlo-based	*HubPPR* [25]	Relative	Medium
		FORA+ [28]	Relative	Fast
Index-free	Iterative-based	*Power* [20]	Additive	Slow
	Local update	*Forward Search* [2]	Not given	Fast
	Matrix-based	*Inverse* [23]	Exact	Slow
	Monte-Carlo-based	*Random Walk Sampling* [9]	Relative	Slow
		BiPPR [17]	Relative	Medium
		TopPPR [29]	Additive	Medium
		FORA [28]	Relative	Medium
		***ResAcc* (ours)**	**Relative**	**Fast**

where p_f is the failure probability. (3) Thirdly, we conducted comprehensive experiments on real datasets containing up to billions of edges. The results demonstrate that *ResAcc* outperforms all the existing algorithms by up to 4 times in terms of query time and by up to 6 orders of magnitude in terms of accuracy, which is considered as a substantial improvement. In particular, our experiments show that the best-known algorithm *FORA* ran nearly 1000 seconds on Twitter, but *RecAcc* ran less than 275 seconds with the same theoretical accuracy. (4) Fourthly, to show the superiority of *ResAcc* over the existing algorithms in real-world applications, we conducted an experiment for the overlapping community detection. The results about the overlapping community detection demonstrate that our proposed method *ResAcc* took less time cost by up to 1 order of magnitude than *FORA*.

We organize the paper as follows. Section II gives our problem definition, and introduces two basic existing techniques and the state-of-the-art *FORA*. Section III elaborates our proposed method *ResAcc*. Sections IV and V present two techniques used in *ResAcc*. The detailed related work and the experimental results are elaborated in Section VI and Section VII, respectively. Section VIII gives the conclusions.

II. PRELIMINARIES

In Section II-A, we first define our problem. Several important concepts are defined in Section II-B, while the background techniques used by the state-of-the-art appear in Section II-C.

A. Problem Definition

Let $G(V, E)$ be a directed unweighted graph with n nodes and m edges. For an undirected graph, we can convert it to a directed one by treating each edge as two opposite directed edges. Same as [25], [28], [29], [22], [14], [23], we assume that the graph has no self-loop. Given a graph $G(V, E)$ and a source node s, *Random Walk with Restart* (RWR) [23] computes the RWR value of each node in $G(V, E)$ w.r.t s by simulating a number of random walks, where each random walk starts from s, and at each step, it either (i) terminates with α probability, or (ii) moves to an out-neighbour of the current node with $(1-\alpha)$ probability. For each $t \in V$, the RWR value $\pi(s, t)$ of t w.r.t s can be regarded as the *stationary probability* that a random walk from s terminates at t. In this paper, we focus on *the approximate single-source RWR query* (SSRWR).

Definition 1 (Approximate SSRWR)**.** *Given a graph $G(V, E)$, a source node s, a threshold δ, a restart probability α, a relative error ϵ and a fail probability p_f, an approximate SSRWR returns the estimated RWR value $\hat{\pi}(s, t)$ such that for each $t \in V$ whose $\pi(s, t) > \delta$, with at least $1-p_f$ probability,*

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

Personalized PageRank (PPR). Personalized PageRank (PPR) [19] is an extension of RWR, which calculates the relevance of nodes according to a preference distribution for a given source node s. A random walk considered by PPR either jumps to a random node according to this preference (with α probability) or moves to an out-neighbour (with $1-\alpha$ probability) [22]. However, most studies on PPR [17], [25], [28], [4], [1], [23], [11] focus on *the single-source PPR query* (SSPPR), which the random walk jumps to s with α probability, and returns the PPR value of all nodes in the graph w.r.t s. In this case, SSPPR is identical to SSRWR.

B. Concepts and Their Definitions

In this section, we formally define several important terms to be used in our proposed method.

Definition 2 (The shortest distance)**.** *Given two nodes in a graph, namely u and v, the shortest distance from u to v is the length of the shortest path from u to v.*

Definition 3 (The i-hop layer)**.** *Given a node v in a graph, the i-hop layer of v, denoted by $L_{i-hop}(v)$, is the set of nodes whose shortest distance from v is exactly i. Besides, when $i = 0$, $L_{0-hop}(v) = \{v\}$.*

Definition 4 (The i-hop set)**.** *Given a node v in a graph, the i-hop set of v, denoted by $V_{i-hop}(v)$, is the set of nodes whose shortest distance from v is at most i. That is, $V_{i-hop}(v) = L_{0-hop}(v) \cup L_{1-hop}(v) \cup ... \cup L_{i-hop}(v)$.*

Definition 5 (The i-hop induced subgraph)**.** *Given a node v in a graph G, the i-hop induced subgraph of v, denoted by $G'_{i-hop}(v)$, is the subgraph of G induced by the i-hop set of v, i.e., $V_{i-hop}(v)$, such that the set of nodes in $G'_{i-hop}(v)$ is $V_{i-hop}(v)$ and the set of edges in $G'_{i-hop}(v)$ is $\{(u, w)|u, w \in V_{i-hop}(v)$ and $(u, w) \in E\}$.*

C. Basic Techniques and The State-Of-The-Art

Next, we introduce two basic techniques for SSRWR, namely *Random Walk sampling* [9] and *Forward Search* [2], which are used in the state-of-the art, *FORA* [28].

Random Walk sampling [9]. Given a source node s, random walk sampling first generates a number of random walks from s and for each $t \in V$, it uses the fraction of walks that terminate at t as an estimation of $\pi(s, t)$, denoted by $\hat{\pi}(s, t)$. Its time cost depends on the number of walks it generates.

Algorithm 1 Forward search

Input: A graph $G(V, E)$, a source node s, the restart probability α, and the residue threshold r^f_{max}
Output: Reserve $\pi^f(s,t)$ and residue $r^f(s,t)$ for each $t \in V$
1: $\pi^f(s,t) \leftarrow 0$ for all $t \in V$;
2: $r^f(s,s) \leftarrow 1$; $r^f(s,t) \leftarrow 0$ for each $t \in V \backslash \{s\}$;
3: **while** $\exists t \in V$ such that $r^f(s,t)/d_{out}(t) \geq r^f_{max}$ **do**
4: Do a forward push operation at node t;
5: **Return** $\pi^f(s,t)$ and $r^f(s,t)$ for each $t \in V$;

According to [9], to guarantee a relative error ϵ, it needs to generate $O(\frac{n \log(n)}{\epsilon^2})$ random walks. Thus, it takes $O(\frac{n \log n}{\alpha \epsilon^2})$ query time since the expected length of a walk is $\frac{1}{\alpha}$, which is expensive for large-scale graphs.

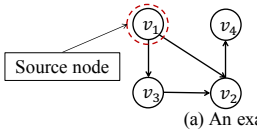
Forward Search [2]. *Forward Search* is a local update algorithm which approximates the RWR value of each node w.r.t a source node s via a graph traversal. Specifically, for each $t \in V$, it maintains a *forward reserve* $\pi^f(s,t)$ and a *forward residue* $r^f(s,t)$ and continually updates them using *forward push operations* (to be defined shortly). Intuitively, the forward residue $r^f(s,t)$ "temporarily" stores some RWR values that belong to t and its out-neighbours. The forward push operation "pushes" the current residue held by t to itself and its out-neighbours denoted by $\mathcal{N}^{out}(t)$. When it finishes, the final *forward reserve* $\pi^f(s,t)$ is an approximate RWR value $\hat{\pi}(s,t)$. Formally, the *push condition* and the *forward push operation* are defined below.

Definition 6 (The push condition)**.** *Given a residue threshold r^f_{max}, a node $t \in V$ is said to satisfy the* push condition *if and only if its residue $r^f(s,t)$ divided by its out-degree $d_{out}(t)$ is at least r^f_{max}, i.e., $\frac{r^f(s,t)}{d_{out}(t)} \geq r^f_{max}$.*

Definition 7 (Forward push operation)**.** *If a node $t \in V$ satisfies the push condition, a* forward push operation *at node t will be performed by executing three actions sequentially: (i) it increases t's reserve $\pi^f(s,t)$ by $\alpha \cdot r^f(s,t)$; (ii) it increases the residue of each out-neighbour of t by $\frac{1-\alpha}{d_{out}(t)} \cdot r^f(s,t)$; and (iii) it sets $r^f(s,t) = 0$.*

Algorithm 1 gives the pseudo-code of *Forward Search*. It is proven in [2] that *Forward Search* takes $O(\frac{1}{\alpha r^f_{max}})$ query time. Given a smaller r^f_{max}, *Forward Search* is slower since it needs to perform more push operations. Besides, for any fixed $r^f_{max} > 0$, *Forward Search* cannot provide any output bound.

FORA [28]. To our best knowledge, *FORA* is the state-of-the-art index-free algorithm for SSRWR, whose key idea is to combine Forward Search and Random Walk sampling. Specifically, *FORA* first performs Forward Search with early termination (using a larger residue threshold r^f_{max}), and subsequently runs a certain number of random walks only from the nodes whose residue is non-zero. In this way, the number of walks required for satisfying the given accuracy is reduced compared with the traditional random walk sampling. To satisfy Equation (1) in Definition 1, *FORA* requires $O(\frac{1}{\alpha \cdot r^f_{max}} + \frac{m \cdot r^f_{max} \cdot c}{\alpha})$ query time where $c = \frac{(2\epsilon/3+2) \cdot \log(2/p_f)}{\epsilon^2 \cdot \delta}$, since it takes $O(\frac{1}{\alpha \cdot r^f_{max}})$ time for Forward Search and generates $O(m \cdot r^f_{max} \cdot c)$ random walks. However, *FORA* is still inefficient (to be elaborated in Section III and Section IV).



Node v	v_1	v_2	v_3	v_4
Initial $r^f(v_1,v)$	1	0	0	0

(a) An example of graph.

	Push operation at node v	$r^f(v_1,v)$			
		v_1	v_2	v_3	v_4
(1)	v_1	0	0.4	0.4	0
(2)	v_2	0	0	0.4	0.32
(3)	v_3	0	0.32	0	0.32
(4)	v_2	0	0	0	0.576

(b) without residue accumulation at node v_2.

	Push operation at node v	$r^f(v_1,v)$			
		v_1	v_2	v_3	v_4
(1)	v_1	0	0.4	0.4	0
(2)	v_3	0	0.72	0	0
(3)	v_2	0	0	0	0.576

(c) with residue accumulation at node v_2.

Fig. 1. Running example of the effect of *residue accumulation*. For each push operation, the *updated residues* are highlighted in *grey*.

III. ResAcc: Residue-Accumulated Approach

In this section, we present our **Residue-Accumulated** approach (*ResAcc*) for SSRWR query. As a whole, *ResAcc* estimates the RWR value $\pi(s,t)$ of each node $t \in V$ w.r.t a source s by applying the following invariant from [1], [28]:

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

where $\pi^f(s,t)$ (resp. $r^f(s,v)$) is the reserve of node t (resp. the residue of node v) w.r.t s. This equation provides a way to compute $\pi(s,t)$ by utilizing the reserves and the residues of all nodes in the graph. However, it is very expensive to compute the RWR value $\pi(v,t)$ for each $v \in V$. To speed up the computation, a rough approximation of $\pi(v,t)$, denoted as $\pi^o(v,t)$, can be computed by utilizing Random Walk sampling so that *ResAcc* estimates the RWR value $\pi(s,t)$ as follows:

$$\hat{\pi}(s,t) = \pi^f(s,t) + \sum_{v \in V} r^f(s,v) \cdot \pi^o(v,t), \quad (3)$$

where $\hat{\pi}(s,t)$ is the estimation of $\pi(s,t)$.

Main challenge. A straightforward solution by exploiting Equation (3) is to first perform Forward Search with a residue threshold r^f_{max} and then simulate the random walks from each node v whose residue $r^f(s,v)$ is non-zero, which is the major idea of *FORA*. However, this solution suffers from low-efficiency issue due to two reasons: (1) Forward Search is inefficient even with a large r^f_{max}, and (2) it requires to simulate a huge number of random walks. In particular, the number of random walks required by *FORA* is proportional to the sum of non-zero residues of all the nodes in the graph, denoted as r_{sum} where $r_{sum} = \sum_{v \in V} r^f(s,v)$, which is usually large due to the large residue threshold r^f_{max}. Thus, the existing technique Forward Search significantly limits the efficiency for computing the reserves and residues, leading to that *FORA* cannot answer SSRWR query efficiently.

A. Intuition of Residue Accumulation

For Algorithm 1 (*Forward Search*), given the source node s, the residue $r^f(s,t)$ of node $t \in V$ can be regarded as a "temporary container" that contains a part of reserves that belong to t's out-neighbours and t itself. Thus, a *forward push operation* at node t can be regarded as a *settlement* to let $r^f(s,t)$ be 0 (since the graph has no self-loop). If we do not perform a push operation at t, $r^f(s,t)$ will increase by receiving the

residues from its in-neighbours and be *accumulated* to a *large* value. This large accumulated residue $r^f(s,t)$ is *important* since node t can perform the push operation only once rather than every time t satisfies the *push condition*. We denote this phenomenon of accumulating large residue values as *residue accumulation*. To illustrate the effect of *residue accumulation*, an running example is given in Figure 1, where Figure 1(a) shows the graph, Figure 1(b) and Figure 1(c) show the push operations *without* and *with* applying the residue accumulation at node v_2, respectively. Specifically, with applying the residue accumulation at v_2, we do not perform the push operation at v_2 until its residue remains unchanged. By comparing Figure 1(b) with Figure 1(c), we can see that the residue accumulation at node v_2 can reduce the total number of push operations of Forward Search from 4 to 3, and the final results on both cases are the same. Although the performance gain in this example is only 1 since the graph is very simple, the gain in the real-world would be very large where the number of in-neighbours of a node is large. Thus, the residue accumulation is very useful to accelerate the computations of SSRWR.

B. Overview of ResAcc

To efficiently solve SSRWR query, *ResAcc* exploits the intuition of *residue accumulation* in a non-trivial way so that it has the following two achievements: (i) it quickly updates the reserves and residues of all nodes in the graph by taking a small amount of time cost, and (ii) the number of random walks required is significantly reduced since r_{sum} is largely reduced. Thus, *ResAcc* is of high efficiency while guaranteeing high accuracy of the estimated RWR values. Towards this end, *ResAcc* computes the reserve and residue of each node by subsequently using two efficient and novel techniques proposed by us, called the h-Hop forward search (h-**HopFWD**) and the one-more forward search (**OMFWD**). As illustrated in Figure 2, *ResAcc* consists of three phases:

- **the h-*HopFWD* phase** (see Section IV for the details). In this phase, *ResAcc* runs h-HopFWD from the source node s by focusing on the h-hop induced subgraph of s (i.e., $G'_{h-hop}(s)$). This phase quickly computes the reserve and residue of each node in $G'_{h-hop}(s)$ (instead of all nodes in the graph) by utilizing the intuition of *residue accumulation*.
- **the *OMFWD* phase** (see Section V for the details). In this phase, *ResAcc* runs *OMFWD* to obtain the final reserve $\pi^f(s,t)$ and final residue $r^f(s,t)$ of each $t \in V$. This phase further reduces r_{sum} quickly due to the *residue accumulation* at the nodes in $L_{(h+1)-hop}(s)$.
- **the *Remedy* phase**. In this phase, *ResAcc* estimates $\hat{\pi}(s,t)$ for each node $t \in V$ by combining Random Walk sampling with the final reserves and residues based on Equation (3).

C. Implementation Details of ResAcc

Algorithm 2 gives the pseudo-code of *ResAcc*. For readability of this algorithm, we regard h-HopFWD and OMFWD as blackboxes here (to be introduced later). *ResAcc* takes as inputs a graph $G(V,E)$, a source node s, a restart probability

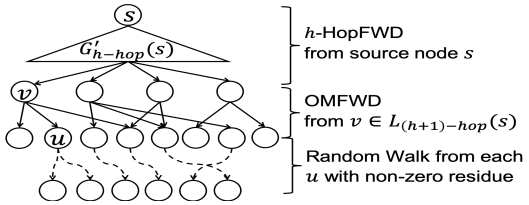


Fig. 2. Illustration of *ResAcc* from the source node s.

α, a residue threshold for h-HopFWD r^{hop}_{max}, a residue threshold for OMFWD r^f_{max}, and the number of hops h. The goal of *ResAcc* is to return the estimated RWR value $\hat{\pi}(s,t)$ of each node t in the graph. Specifically, *ResAcc* first initializes the estimated RWR value $\hat{\pi}(s,t) = 0$ for each node $t \in V$ and the forward residue $r^f(s,t)$ such that $r^f(s,s) = 1$ and $r^f(s,t) = 0$ for each $t \in V$ where $t \neq s$ (Lines 1-2). Then, it starts the h-HopFWD phase by invoking Algorithm 3 (to be introduced later) by taking as inputs the source s, the threshold r^{hop}_{max}, parameter h, and the current reserve and residue of each node (Line 3). Next, it starts the OMFWD phase by invoking Algorithm 4 (to be introduced later) taking as input r^f_{max} and the current reserve and residue of each node (Line 4). After that, the reserve $\hat{\pi}(s,t)$ and residue $r^f(s,t)$ of each $t \in V$ are obtained. Finally, it starts the remedy phase (Lines 5-17).

In the remedy phase, *ResAcc* estimates $\sum_{v \in V} r^f(s,v) \cdot \pi^o(v,t)$ in Equation (3) by simulating a number of random walks from each v whose residue $r^f(s,v)$ is non-zero. Specifically, it computes the total residue of all nodes r_{sum}, based on which it derives a value n_r that will be used to decide the number of random walks from each node v (Line 6-7). After that, it proceeds to estimate $r^f(s,v) \cdot \pi^o(v,t)$ for each v whose residues are larger than zero (Lines 8-15). In particular, for each $t \in V$, it initializes a value C_t to be zero (i.e., $C_t = 0$), where C_t is the estimated value of $\sum_{v \in V} r^f(s,v) \cdot \pi^o(v,t)$ (Line 8). After that, for each node v, it performs $n_r(v)$ random walks from v, where $n_r(v)$ is defined as below:

$$n_r(v) = \left\lceil \frac{r^f(s,v) \cdot n_r}{r_{sum}} \right\rceil.$$

If a random walk terminates at a node t, then *ResAcc* increases C_t by $\frac{a(v) \cdot r_{sum}}{n_r}$, where $a(v) = \frac{r^f(s,v)}{r_{sum}} \cdot \frac{n_r}{n_r(v)}$ (Lines 11-15). After each v whose residue is non-zero is processed, for each $t \in V$, the algorithm increases $\hat{\pi}(s,t)$ by C_t (Lines 16-17). Then, the algorithm terminates.

D. Accuracy Guarantee

We first prove that the results returned by *ResAcc* are unbiased. Due to space limit, we give the proof sketches of all lemmas and theorems below while the step-by-step proofs could be found in the appendix (from Appendix M to Appendix O).

Theorem 1. *The expectation of $\hat{\pi}(s,t)$ returned by Algorithm 2 is equal to $\pi(s,t)$, i.e., $E[\hat{\pi}(s,t)] = \pi(s,t)$.*

Proof. Firstly, to prove this theorem, it is equal to prove that $E[C_t] = \sum_{v \in V} r^f(s,v) \cdot \pi(v,t)$. Next, we prove that when *ResAcc* processes a node v whose residue is non-zero, the expected amount of increment of C_t is exactly $r^f(s,v) \cdot \pi(v,t)$

Algorithm 2 ResAcc

Input: A graph $G(V, E)$, the source node s, the restart probability α, residue thresholds r^{hop}_{max} and r^{f}_{max}, and the number of hops h
Output: Reserve $\hat{\pi}(s,t)$ for each $t \in V$

$\hat{\pi}(s,t) \leftarrow 0$ for all $t \in V$;
$r^f(s,s) \leftarrow 1$; $r^f(s,t) \leftarrow 0$ for each $t \in V$ such that $t \neq s$;
$[\hat{\pi}_s, r^f_s] \leftarrow h$-HopFWD$(s, r^{hop}_{max}, h, [\hat{\pi}_s, r^f_s])$;
$[\hat{\pi}_s, r^f_s] \leftarrow$ OMFWD$(r^f_{max}, [\hat{\pi}_s, r^f_s])$;
/*Start the remedy process*/
Compute $r_{sum} = \sum_{v \in V} r^f(s,v)$;
Compute $n_r = r_{sum} \cdot \frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2 \cdot \delta}$;
$C_t \leftarrow 0$ for each $t \in V$;
for $v \in V$ with $r^f(s,v) > 0$ **do**
  Let $n_r(v) = \left\lceil \frac{r^f(s,v)\cdot n_r}{r_{sum}} \right\rceil$;
  Let $a(v) = \frac{r^f(s,v)}{r_{sum}} \cdot \frac{n_r}{n_r(v)}$;
  for $i = 1$ to $n_r(v)$ **do**
    Generate a random walk from v;
    Let t be the last node of this walk;
    $C_t \leftarrow C_t + \frac{a(v)\cdot r_{sum}}{n_r}$;
for each node $t \in V$ **do**
  $\hat{\pi}(s,t) \leftarrow \hat{\pi}(s,t) + C_t$;
Return $\hat{\pi}(s,t)$ for each $t \in V$;

based on the definition of $n_r, n_r(v)$, and $a(v)$. Finally, by processing all nodes, $E[C_t] = \sum_{v \in V} r^f(s,v) \cdot \pi(v,t)$. □

Next, we show that *ResAcc* guarantees the accuracy of the estimated RWR values by applying the following concentration bound as shown in Theorem 2 from [28].

Theorem 2 ([28]). *Let $X_1, ..., X_{n_r}$ be independent random variables with $Pr[X_i = 1] = p_i$ and $Pr[X_i = 0] = 1 - p_i$. Let $X = \frac{1}{n_r}\sum_{i=1}^{n_r} a_i X_i$ with $a_i > 0$, and $\zeta = \frac{1}{n_r}\sum_{i=1}^{n_r} a_i^2 \cdot p_i$. By letting $a = \max\{a_1, ..., a_{n_r}\}$, the following inequality holds:*

$$Pr[|X - E[X]| \geq \lambda] \leq 2 \cdot \exp(-\frac{\lambda^2 \cdot n_r}{2\zeta + 2a\lambda/3}).$$

Lemma 1. *For any node t, given an arbitrary relative error ϵ, we have the following inequality: $Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon \cdot \pi(s,t)] \leq 2 \cdot \exp(-\frac{\epsilon^2 \cdot n_r \cdot \pi(s,t)}{r_{sum} \cdot (2+2\epsilon/3)})$.*

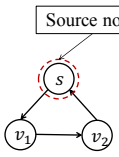
Proof. Firstly, we define some notations. Let $b_j = a(v)$ if the j-random walk starts from a node $v \in V$ where $j \in \{1, ..., n_r\}$. We can know that $max_j b_j = 1$, and $b_j^2 \leq b_j$ for any j since $a(v) \leq 1$. Then, we define $Y_j(t)$ be the random variable such that:

$$Y_j(t) = \begin{cases} 1, & \text{if the } j\text{-th walk ends at } t, \\ 0, & \text{otherwise.} \end{cases}$$

Let $Y = \frac{1}{n_r}\sum_{j=1}^{n_r} b_j Y_j(t)$, and $\zeta = \frac{1}{n_r}\sum_{j=1}^{n_r} b_i^2 \cdot E[Y_j(t)]$. Let $a = \max\{b_1, ..., b_{n_r}\}$. By definition, $b_j^2 \leq 1$, and so, $\zeta \leq E[Y]$ and $a \leq 1$. Secondly, by substituting ζ and a in Theorem 2, we have that: $Pr[|Y - E[Y]| \geq \lambda] \leq 2 \cdot \exp(-\frac{\lambda^2 \cdot n_r}{2E[Y] + 2\lambda/3})$. Since $\pi(s,t) - \hat{\pi}(s,t) = \frac{n_r(v)\cdot r_{sum}}{n_r}(E[Y] - Y)$, we have $Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \frac{n_r(v)\cdot r_{sum}}{n_r} \cdot \lambda] \leq 2\exp(-\frac{\lambda^2 \cdot n_r}{2E[Y] + 2\lambda/3})$. Finally, we complete the proof due to the facts that $E[Y] \leq \frac{n_r}{n_r(v)\cdot r_{sum}} \cdot \pi(s,t)$, $\lambda = \epsilon \cdot \frac{n_r \cdot \pi(s,t)}{n_r(v)\cdot r_{sum}}$ and $a < 1$. □

Theorem 3. *For any node t with $\pi(s,t) > \delta$, if $n_r \geq r_{sum} \cdot \frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2 \cdot \delta}$, ResAcc returns an approximate RWR $\hat{\pi}(s,t)$ that satisfies Equation(1) with at least $1 - p_f$ probability.*

Proof. We prove that $Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon \cdot \pi(s,t)] \leq p_f$ by substituting n_r and $\pi(s,t)$ in Lemma 1. □



(a) A graph for example.

	Push operation at node v	$r^f_0(s,v)$			$\pi^f_0(s,v)$		
		s	v_1	v_2	s	v_1	v_2
Initial	None	1	0	0	0	0	0
(1)	s	0	0.8	0	0.2	0	0
(2)	v_1	0	0	0.64	0.2	0.16	0
(3)	v_2	0.512	0	0	0.2	0.16	0.128

(b) Subsequent forward push operations with the initial residue of s, $r^f_0(s,v) = 1$.

	Push operation at node v	$r^f_1(s,v)$			$\pi^f_1(s,v)$		
		s	v_1	v_2	s	v_1	v_2
Initial	None	0.512	0	0	0	0	0
(1)	s	0	0.4096	0	0.1024	0	0
(2)	v_1	0	0	0.32768	0.1024	0.08192	0
(3)	v_2	0.262144	0	0	0.1024	0.08192	0.065536

(c) Subsequent forward push operations with the initial residue of s, $r^f_1(s,v) = 0.512$.

Fig. 3. Running example of the *looping* phenomenon where the restart probability $\alpha = 0.2$ and the residue threshold $r^f_{max} = 0.1$. For each push operation, the newly updated residue and reserve are highlighted in *grey*.

IV. New Technique: h-HopFWD

A. Observation: Looping Phenomenon

We observed that the *Forward Search* (Algorithm 1) has the *looping* phenomenon at the source node s. Initially, *Forward Search* assigns 1 to the residue of s w.r.t s, i.e., $r^f(s,s) = 1$. For simplicity, we denote this initial residue of s as $r^f_0(s,s)$. Subsequently, *Forward Search* performs the very first *forward push operation* at node s since only s satisfies the push condition (while currently other nodes have zero residue), after which, the residue of s becomes zero. However, during the remaining process of *Forward Search*, the residue of s might become *non-zero* again via its in-neighbours, denoted by $r^f_1(s,s)$ to differentiate from $r^f_0(s,s)$. Since $r^f_1(s,s)$ is non-zero, the algorithm needs to do another forward push operation at s again (if it satisfies the push condition). However, we observed that all the operations done with the originally residue value (i.e., $r^f_0(s,s) = 1$) have to be repeated with the newly updated residue $r^f_1(s,s)$.

To illustrate, a running example is given in Figure 3 where Figure 3(a) shows the graph. In particular, Figure 3(b) illustrates three push operations performed when the initial residue of s ($r^f_0(s,s)$) is set to be 1. Specifically, the algorithm subsequently performs a push operation at s, v_1, and v_2. For each push operation, the newly updated residue and reserve are highlighted in *grey*. We can see that after the third push operation (at node v_2), the residue of s becomes non-zero (i.e., 0.512), which consequentially leads the algorithm perform a push operation at s again. Next, Figure 3(c) illustrates the three push operations performed when the initial residue of s becomes 0.512. However, the orderings of push operations performed at this time is the same as in Figure 3(b). Thus, a *looping* phenomenon exists at node s, leading to the *redundant* operations in *Forward Search* since such *loopings* at node s will continue to happen until the final residue of s cannot satisfy the push condition. For example, the final residue of s in Figure 3(c) is 0.262144 ($> r^f_{max} = 0.1$), which makes another loop at s, leading to low-efficiency.

B. Details of h-HopFWD

To avoid the *looping* phenomenon in *Forward Search*, we propose a new technique called h-HopFWD by *accumulating the residue* of the source node s so that the *looping* at s is cut down to avoid the redundant push operations. However, it is computationally expensive to accumulate the residue of s in the whole graph since it takes large time cost to let all the nodes in the graph except s not satisfy the push condition. To address this issue, h-HopFWD exploits the *hop-based induced subgraph* constructed from source s, namely $G'_{h-hop}(s)$ such that it can perform the push operations at only the nodes in $G'_{h-hop}(s)$, instead of at all the nodes in the graph, and so its time cost is extremely low. Besides, the subgraph helps to accumulate the residue of nodes in $L_{(h+1)-hop}(s)$ to be a large value (see Section V). As a whole, h-HopFWD has two phases: the *accumulating* phase and the *updating* phase. In the *accumulating* phase, it accumulates the residue of s after the first push operation. This phase continues until the residue of s remains unchanged. In the *updating* phase, it computes the reserve and residue of each node in the subgraph in $O(1)$ time by utilizing the accumulated residue of s.

The updating phase. However, it comes a question: *how to compute the reserve and residue of each node in the subgraph?* The updating phase is based on Lemma 2, which indicates that the ordering of all the push operations done with $r_0^f(s,s)=1$ (the original residue of s) could be the same as these with the accumulated residue $r_1^f(s,s)$ after the accumulating phase (if it is not zero) *by adjusting the push condition.*

Lemma 2. *Given a residue threshold r_{max}^{hop}, the ordering of all the push operations done with $r_0^f(s,s)=1$ could be* identical *to these with $r_1^f(s,s)$ by changing the push condition with $r_1^f(s,s)$ as follows: a node $t \in V$ is said to satisfy the push condition if and only if its residue divided by its out-degree $d_{out}(t)$ is at least $r_{max}^{hop} \cdot r_1^f(s,s)$ (instead of r_{max}^{hop} as previous).*

Proof. For the case with $r_0^f(s,s) = 1$, we assume that the total number of push operations is l. We denote the ordering of nodes selected for the push operations as $\{N\}_l = \{v_1, v_2, ..., v_l\}$ where $v_i \in V$. Similarly, We denote the ordering of nodes with $r_1^f(s,s)$ as $\{N'\}_{l'} = \{v_1^{'}, v_2^{'}, ..., v_{l'}^{'}\}$ where l' is the number of push operations done with $r_1^f(s,s)$. Using Mathematical Induction, we prove that $\{N\}_l$ is equal to $\{N'\}_{l'}$ such that: (i) $l = l'$ and (ii) $v_i = v_i^{'}$ for each i. □

Besides, we observe that for a fixed initial residue of the source node s, says $r_0^f(s,s)$, in the *accumulating* phase, the reserve and residue of each node are proportional to $r_0^f(s,s)$ with different coefficients. For example, given a graph shown in Figure 3(a), with $r_0^f(s,s)$, the residue of node v_1 is equal to $\frac{(1-\alpha)}{d_{out}(s)} \cdot r_0^f(s,s)$ by a push operation at node s; and the residue of node v_2 is equal to $\frac{(1-\alpha)r_0^f(s,v_1)}{d_{out}(v_1)}$, which is $\frac{(1-\alpha)^2}{d_{out}(v_1)d_{out}(s)} \cdot r_0^f(s,s)$, by a push operation at node v_1. Thus, if we know the reserve and residue of any node $t \in V$ after the *accumulating* phase with $r_0^f(s,s)$, it is easy to know the reserve and residue of node t after the *accumulating* phase with a different value for the initial residue of s, says $r_1^f(s,s)$, since the ordering of push operations with $r_1^f(s,s)$ is the same as previous (Lemma 2). To illustrate, we denote the *accumulating* phase with $r_0^f(s,s)=1$ and $r_1^f(s,s)$ as *Phase-1* and *Phase-2*, respectively. Let $\pi_1^f(s,t)$ and $r_1^f(s,t)$ be the reserve and residue of any node $t \in V$ after *Phase-1*, respectively. Let $\pi_2^f(s,t)$ and $r_2^f(s,t)$ be the reserve and residue of any node $t \in V$ after *Phase-2*, respectively. For any node $t \in V$, we can derive a relationship between $\pi_1^f(s,t)$ and $\pi_2^f(s,t)$, and a relationship between $r_1^f(s,t)$ and $r_2^f(s,t)$ as follows:

$$\frac{\pi_2^f(s,t)}{r_1^f(s,s)} = \frac{\pi_1^f(s,t)}{r_0^f(s,s)} \text{ and } \frac{r_2^f(s,t)}{r_1^f(s,s)} = \frac{r_1^f(s,t)}{r_0^f(s,s)},$$

which could be verified by the example in Figure 3.

Moreover, we observe that if the residue of s obtained by *Phase-2* is larger than the residue threshold r_{max}^{hop}, another *accumulating* phase could be triggered. Let T denote the total number of the *accumulating* phases with a given residue threshold r_{max}^{hop}. For any node $t \in V$, by summing up the reserves (or residues) of t in all T *accumulating* phases, we can obtained the final reserve (residue) of t w.r.t s. Instead of generating T accumulating phases one by one, the *updating* phase of h-HopFWD exploits the relationships between the reserve (or residue) of t obtained after the i-th accumulating phase, denoted as $\pi_i^f(s,t)$ (or $r_i^f(s,t)$), and the reserve π_1^f (or the residue $r_1^f(s,t)$):

$$\frac{\pi_i^f(s,t)}{r_{i-1}^f(s,s)} = \frac{\pi_1^f(s,t)}{r_0^f(s,s)} \text{ and } \frac{r_i^f(s,t)}{r_{i-1}^f(s,s)} = \frac{r_1^f(s,t)}{r_0^f(s,s)}.$$

It also indicates that $r_i^f(s,s) = [r_1^f(s,s)]^i$. Thus, the updating phase computes the final reserve and residue of t in $O(1)$ time, leading to high efficiency. Specifically, in the updating phase, if $r_1^f(s,s)$ is not equal to zero, it computes the reserve and residue of each node $t \in V$ as follows:

$$\pi^f(s,t) = \pi_1^f(s,t) \cdot S \tag{4}$$

Algorithm 3 h-HopFWD

Input: Graph $G(V,E)$, source node s, restart probability α, residue threshold r_{max}^{hop}, the number of hops h, reserve $\pi^f(s,t)$ and residue $r^f(s,t)$ of each node $t \in V$
Output: Reserve $\pi^f(s,t)$ for each $t \in V_{h-hop}(s)$ and residue $r^f(s,t)$ for each $t \in V_{h-hop}(s) \cup L_{(h+1)-hop}(s)$

/*Start the accumulating phase*/
Perform a single forward push operation at s;
while $\exists t \in V_{h-hop}(s) \backslash \{s\}$ such that $\frac{r^f(s,t)}{d_{out}(t)} \geq r_{max}^{hop}$ **do**
  for each $v \in \mathcal{N}^{out}(t)$ **do**
    $r^f(s,v) \leftarrow r^f(s,v) + (1-\alpha) \cdot \frac{r^f(s,t)}{d_{out}(t)}$;
  $\pi^f(s,t) \leftarrow \pi^f(s,t) + \alpha \cdot r^f(s,t)$;
  $r^f(s,t) \leftarrow 0$;
/*Start the updating phase*/
$T \leftarrow \left\lceil \frac{\log(r_{max}^{hop} \cdot d_{out}(s))}{\log r^f(s,s)} \right\rceil$; //compute the maximum number of loops at s
$S \leftarrow \frac{1-[r^f(s,s)]^{T-1}}{1-r^f(s,s)}$; //compute the scaler
for each $v \in V_{h-hop}(s)$ **do**
  $\pi^f(s,v) \leftarrow \pi^f(s,v) \cdot S$;
  if v is the source node s **then**
    $r^f(s,v) \leftarrow [r^f(s,v)]^T$;
  else
    $r^f(s,v) \leftarrow r^f(s,v) \cdot S$;
for each $v \in L_{(h+1)-hop}(s)$ **do**
  $r^f(s,v) \leftarrow r^f(s,v) \cdot S$;
Return Reserve $\pi^f(s,t)$ for each $t \in V_{h-hop}(s)$ and residue $r^f(s,t)$ for each $t \in V_{h-hop}(s) \cup L_{(h+1)-hop}(s)$;

$$r^f(s,t) = \begin{cases} r_1^f(s,t) \cdot S & \text{, if } t \neq s \\ [r_1^f(s,t)]^T & \text{, otherwise} \end{cases} \quad (5)$$

where $S = \frac{1-[r_1^f(s,s)]^{T-1}}{1-r_1^f(s,s)}$, $T = \left\lceil \frac{\log[r_{max}^{hop} \cdot d_{out}(s)]}{\log r_1^f(s,s)} \right\rceil$.
Algorithm 3 gives the pseudo-code of h-HopFWD. Lemma 3 shows that h-HopFWD computes the reserve and residue of each node in the subgraph correctly (whose step-by-step proof could be found in Appendix Q).

Lemma 3. *If $r_1^f(s,s) \neq 0$, the reserve and residue of any node t by h-HopFWD are correct. Besides, $r^f(s,s) < r_{max}^{hop} \cdot d_{out}(s)$.*

Proof. We prove this using the relationships stated above. □

Next, we bound the sum of non-zero residues of all nodes obtained by h-HopFWD, denoted by r_{sum}^{hop}, in Lemma 4. Its step-by-step proof could be found in Appendix R.

Lemma 4. *If r_{max}^{hop} is small enough such that each node $v \in V_{h-hop}(s)$ performs at least one push operation, then r_{sum}^{hop} is bounded where $r_{sum}^{hop} \leq (1-\alpha)^h$.*

Proof. From the definition of RWR, we have the invariant that $r_{sum}^{hop} + \sum_{v \in V_{h-hop}(s)} \pi^f(s,v) = 1$ after h-HopFWD terminates. Based on this, we prove that r_{sum}^{hop} is largest if h-HopFWD performs only one push operation at each $v \in V_{h-hop}(s)$, by which we can compute the residues of nodes in $L_{j-hop}(s)$ for each $0 \leq j \leq h$. By summing up those resides, we prove that r_{sum}^{hop} is at most $(1-\alpha)^h$. □

V. Another Technique: OMFWD

In the h-HopFWD phase, the push operations performed at the nodes in the last layer of the h-hop subgraph (i.e., $L_{h-hop}(s)$) are "special" since they push the residues to the nodes which are not in the subgraph, namely the nodes in $L_{(h+1)-hop}(s)$. As defined, the nodes in $L_{(h+1)-hop}(s)$ cannot perform the push operations even though their residues satisfy the push condition. Thus, the residue of each node in $L_{(h+1)-hop}(s)$ is *accumulated* to a *large* value.

Motivated by this, we propose OMFWD which performs the forward push operations from the nodes with accumulated residues. Algorithm 4 gives its pseudocode. Given a new residue threshold r_{max}^f, which is different from r_{max}^{hop} used in h-hopFWD, OMFWD performs the recursive push operations at the nodes which satisfy the push condition with r_{max}^f. After termination, it returns the updated reserves and residues of all node in the graph. Let r_{sum} denote the sum of all residues after OMFWD finishes. Note that r_{sum} is very smaller, resulting in less random walks in the remedy phase.

Algorithm 4 OMFWD

Input: A graph $G(V,E)$, a source node s, the restart probability α, and the residue threshold r_{max}^f, the current reserve $\pi^f(s,t)$ and residue $r^f(s,t)$ for each $t \in V$, the set $L_{(h+1)-hop}(s)$
Output: Final reserve $\pi^f(s,t)$ and residue $r^f(s,t)$ for each $t \in V$

```
Enqueue each nodes in L_{(h+1)-hop}(s) in the decreasing order of residue;
while the queue is not empty do
    Dequeue a node from queue and set it to be t;
    π^f(s,t) ← π^f(s,t) + α · r^f(s,t);
    for each v ∈ N^{out}(t) do
        r^f(s,v) ← r^f(s,v) + (1−α) · r^f(s,t)/d_{out}(t);
        if r^f(s,v)/d_{out}(v) ≥ r_{max}^f then
            Enqueue node v to the queue;
    r^f(s,t) ← 0;
Return π^f(s,t) and r^f(s,t) for each t ∈ V;
```

VI. Other Related Work

A. Existing Work for SSRWR Query

For completeness, this section includes the existing work for the SSPPR query and discusses how to extend the existing work for the Multiple-Sources RWR (MSRWR) query. The existing approaches could be categorized into four types: (i) the iterative-based approaches, (ii) the local update approaches, (iii) the matrix-based approaches, and (iv) the Monte-Carlo-based approaches. Table I compares them according to the three requirements mentioned in Section 1.

Iterative-based. *Power* [20] is an index-free method which iteratively updates the RWR values of all nodes w.r.t the source until convergence. The time complexity of *Power* is $O(mT)$ since it traverses all edges in the graph in each iteration where T is the number of iterations, which is huge on large graphs and cannot satisfy the high-efficiency requirement. *TPA* [31] is an index-based iterative method. Specifically, in the preprocessing phase, *TPA* estimates the RWR values of nodes far from the source node using their PageRank scores. In the query phase, it estimates RWR values of nodes close to the source node using *Power*. However, the same as *Power*, *TPA* suffers from expensive time cost in the query phase.

Local update. There are two local update approaches in the literature: *Forward Search* [2] (destribed in Section II-C) and *Backward Search* [1], [26], both of which are index-free. Unlike *Forward Search*, *Backward Search* performs a graph traversal from a target node via the reverse direction of edges and returns the approximate RWR values of a target node w.r.t all the nodes in the graph. *Backward Search* is computationally expensive for the SSRWR query since it has to perform backward searches from *each* node in the graph. Besides, both approaches cannot guarantee the result accuracy.

Matrix-based. According to [22], [23], [11], [14], given a source node s, the RWR values of all nodes w.r.t s can be computed to by $\boldsymbol{\pi}_s = \alpha(\mathbf{I} - (1-\alpha) \cdot \mathbf{D}^{-1}\mathbf{A}^T)^{-1}\boldsymbol{e}_s$. Thus, the exact RWR values can be obtained by computing a matrix inversion, which is time-consuming. The existing matrix-based approaches utilize different matrix decompositions in the preprocessing phase to reduce the time for computing a matrix inversion in the query phase. Thus, most of them are index-oriented and do not satisfy the index-free requirement. Frequently-used matrix optimization methods include: low-rank approximation [23], LU decomposition [21], QR decomposition [11], [21], and Complete Schurment [22], [14]. Unfortunately, most of the matrix-based approaches does not meet the high-efficiency requirement since they take $O(n^2)$ query time in the worst case. Besides, some of them cannot provide the accuracy guarantee (e.g., *B-LIN* [23] and *QR* [11]).

Monte-Carlo-based. The Monte-Carlo-based technique is ex-

ploited by *BiPPR* [22], *HubPPR* [25], *TopPPR* [29] and *FORA*/*FORA*+ [28], [27] (described in Section II). Among them, *BiPPR* and *HubPPR* were proposed for the pairwise PPR query, where the goal is to approximate the value $\pi(s,t)$ given a pair of nodes s and t. *BiPPR* is a combination of *Random Walk sampling* and *Backward Search* [1], which first generates a number of random walks from source s, then runs *Backward Search* from target t, and finally estimates $\pi(s,t)$. *HubPPR* is the index-version of *BiPPR*, which stores the results of random walk sampling (and backward search) for some "hub" nodes in the preprocessing phase. However, when being adapted for SSRWR query, both *BiPPR* and *HubPPR* are time-consuming since they have to execute the backward search for each node in the graph. As shown in [28], *FORA* runs faster than *BiPPR* and *HubPPR*, guaranteeing the same relative error. *TopPPR* was proposed for the top-$\mathcal{K}$ PPR query. It combines *Forward Search*, *Backward Search* and *Random walk sampling* to return the top-$\mathcal{K}$ nodes. Although it is index-free and can be adapted for the SSRWR query, it does not satisfy the high-efficiency requirement since it needs to perform the backward search from each node in the graph, which is expensive.

Extension to MSRWR query. Unlike SSRWR query, MSRWR takes as an input a set $\mathcal{S}$ of source nodes, and outputs the RWR scores of each node in the graph w.r.t each source node $s \in \mathcal{S}$. However, to the best of our knowledge, no existing work studies how to solve MSRWR query efficiently. Meanwhile, no existing work conducted the experiments for MSRWR query. We are the first one to conduct the experiments for MSRWR. A natural method to extend the existing methods for MSRWR query is executing them for each node $s \in \mathcal{S}$ by $|\mathcal{S}|$ times where $|\mathcal{S}|$ is the number of nodes in set $\mathcal{S}$. Our experiments show that *ResAcc* is the fastest for answering MSRWR query among all index-free methods. Besides, *ResAcc* achieves the highest empirical accuracy among all methods by up to 9 orders of magnitude.

B. Comparison with Particle Filtering

Particle Filtering (*PF*) [15], [13] is an alternative technique of the Monte-Carlo simulations (i.e. *MC*) by combining a "deterministic" distribution phase and a random sampling phase. Suppose that the total number of random walks to be generated is w. In the "deterministic" distribution phase, *PF* computes a value w_v for each node $v \in V$ where w_v is the number of random walks starting from the source node s visiting v. Specifically, for each node whose w_v divided by its out-degree $d_{out}(v)$ is at least a threshold w_{min} (i.e., $\frac{w_v}{d_{out}(v)} \geq w_{min}$), for each out-neighbour u of v, *PF* "deterministically" increases w_u by $\frac{w_v}{d_{out}(v)}$. But, for each node v whose w_v divided by $d_{out}(v)$ is smaller than w_{min}, *PF* switches to the random sampling phase by randomly selecting an out-neighbour u of v and increasing w_u by w_{min}. This random phase for node v repeats for at most $\lfloor \frac{w_v}{w_{min}} \rfloor$ times. However, *PF* cannot provide the accuracy of estimated RWR values and its "empirical" accuracy is low since its randomized process directly select an out-neighbour of a node based on a user-specified parameter w_{min}. The larger the w_{min}, the larger the error. Our experiments show that *PF* was outperformed by *ResAcc* in terms of accuracy by up to 4 orders of magnitude, running in similar query time.

TABLE II
DATASETS.(K = 10^3, M=10^6, B=10^9)

Dataset	n	m	$\frac{m}{n}$	h
DBLP	317K	2.1M	6.6	3
Web-Stan	282K	2.3M	8.2	2
Pokec	1.63M	30.6M	18.8	2
LJ	4.8M	69.0M	17.4	2
Orkut	3.1M	117.2M	38.1	2
Twitter	41.7M	1.5B	35.3	2
Friendster	65.7M	2.1B	38.1	2

TABLE III
THE AVERAGE QUERY TIME (IN SECONDS) OF EACH INDEX-FREE ALGORITHM FOR SSRWR QUERY VS. DATASET. THE WORD "O.O.T" MEANS THE ALGORITHM RUNS EXCEEDING 1 DAY.

	Power	FWD	MC	FORA	TopPPR	ResAcc
DBLP	76.596	2.60	19.21946	1.091	1.0324	**0.5126**
Web-Stan	0.324	3.904	9.2242	0.182	0.1534	**0.031**
Pokec	733.174	22.400	118.23	13.945	69.4092	**5.6384**
LJ	958.011	45.405	262.54	23.715	78.8589	**11.9546**
Orkut	4452.06	123.715	451.8	596.186	196.211	**23.064**
Twitter	68566.12	720.796	8389.34	979.516	1672.6	**274.722**
Friendster	o.o.t	2863.45	o.o.t	o.o.t	o.o.t	**643.828**

VII. EXPERIMENTS

A. Experimental Setup

All experiments were conducted on a Linux machine with Intel 2.20GHz CPU and 64GB memory. We used 7 real graphs in our experiments: *DBLP, Web-Stan, Pokec, LJ, Orkut, Twitter* and *Friendster*, which are the benchmarks in previous studies [25], [28], [22], [14]. Table II summarizes their statistics. For each dataset, we chose 50 source nodes uniformly at random. An average query time was reported.

We compared our proposed approach, *ResAcc*, against 9 existing algorithms, which can be categorized into two types: index-free approaches and index-oriented approaches. Specifically, the index-free approaches are: (1) *Power*, which generates the ground truth [20], (2) *Forward Search* (FWD) [2], (3) *Random Walk sampling* (MC) [4], (4) *FORA*, which has the best query performance among Monte-Carlo-based algorithms [28], (5) *TopPPR*, which has the best query performance for the top-$\mathcal{K}$ query [29], and (6) *ResAcc*, which is our proposed method. Since *TopPPR* solves the top-$\mathcal{K}$ query, we let $\mathcal{K} = 10^5$ to optimize the performance of *TopPPR* in terms of both the efficiency and accuracy (the effect of $\mathcal{K}$ for *TopPPR* was evaluated in Section VII-F). We did not compare *ResAcc* with other existing index-free methods in Table I since they are empirically outperformed by the above 5 existing methods in [28]. Besides, the index-oriented approaches are: (1) *BePI*, which has the best performance among matrix-based index-oriented algorithms [14], (2) *TPA*, which has the best performance among iterative-based index-oriented algorithms [31], and (3) *FORA*+ [28]. The performance of other index-oriented algorithms introduced in Section VI were dominated by the above approaches as evaluated in [14], [28] and thus, are

TABLE IV
PERFORMANCE OF EACH INDEX-BASED ALGORITHM VS. *ResAcc*.("O.O.M" MEANS "OUT OF MEMORY")

Dataset	Average query time				Preprocessing time				Index size				Graph
	BePI	TPA	FORA+	ResAcc	BePI	TPA	FORA+	ResAcc	BePI	TPA	FORA+	ResAcc	size
DBLP	0.272	3.136	**0.16**	0.5126	4.165	7.31	10.359	**0**	156.1MB	15MB	38.9MB	**0**	18.4MB
Web-Stan	0.09	0.856	0.157	**0.031**	2.550	3.59	3.84	**0**	113.3MB	6.7MB	38.4MB	**0**	10.4MB
Pokec	12.131	38.256	**1.771**	5.6384	65.357	70.96	112.334	**0**	2.65GB	40MB	330MB	**0**	130.8MB
LJ	20.282	85.916	**3.786**	11.9546	140.621	167.6	190.559	**0**	5.088GB	120MB	583MB	**0**	295.2MB
Orkut	o.o.m	140.271	**9.019**	23.064	o.o.m	282.74	453.741	**0**	o.o.m	76MB	879MB	**0**	950.1MB
Twitter	o.o.m	1954.957	**165.715**	274.722	o.o.m	4323.74	5634.31	**0**	o.o.m	1.1GB	12.6GB	**0**	6.2GB
Friendster	o.o.m	o.o.m	o.o.m	**643.828**	o.o.m	o.o.m	o.o.m	**0**	o.o.m	o.o.m	o.o.m	**0**	31GB

excluded. We obtained the codes of *BePI* from [14], *TPA* from [31], *FORA*/*FORA+* from [28] and *TopPPR* from [29]. All algorithms were implemented in C++ except *BePI* and *TPA* implemented in both C++ and Matlab (due to the matrix operation library usage).

For all methods, $\alpha = 0.2$ following previous work [25], [28], [29], [2], [31]. For fair comparison, we set the parameters of each approach mainly following the best setting reported in [17], [28], [14], [25]. Specifically, we set r^f_{max} to be 10^{-12} in *FWD* and we tune the hub selection ratio in *BePI* so that its efficiency is maximized on each dataset. In *MC*, *FORA*, *FORA+* and *ResAcc*, we set $\delta = 1/n$, $p_f = 1/n$, and $\epsilon = 0.5$. Moreover, in *ResAcc*, we set $r^f_{max} = \frac{1}{10 \cdot m}$, $r^{hop}_{max} = 10^{-14}$. The value of h for each dataset is as shown in the last column of Table II. The effect of parameter h and r^{hop}_{max} are evaluated in Appendix G and Section VII-G, respectively.

Following [29], we evaluated the accuracy of each method using two classic metrics: *absolute error* and *Normalized Discounted Cumulative Gain* (NDCG). Detailed description of NDCG could be found in [29].

B. Experimental Results for SSRWR Query

1) Query Time: **Index-free approaches.** Table III shows the query time of the index-free approaches. From Table III, we observe that *ResAcc* takes the least time consistently in all cases. For example, on Twitter, *ResAcc* is 250 times faster than *Power* and is around 3 times faster than *FWD*, *FORA* and *TopPPR*. In particular, compared with *FORA* (the state-of-the-art), *ResAcc* is at least 2 times faster on most datasets. it clearly demonstrates that *ResAcc* satisfies the high-efficiency requirement even on large-scale graphs.

Index-oriented approaches. Table IV compares *ResAcc* against the index-oriented approaches by measuring the query time, the preprocessing time and the index size for each approach. Note that *ResAcc* is index-free and so, it has **zero** preprocessing time and index size. However, it is compared in this experiment to verify that even without indexing, *ResAcc* can achieve a comparable query time as the index-oriented approaches. and meanwhile, it gets rid of the large preprocessing time and high space overhead, making it suitable for supporting online SSRWR. Thus, *ResAcc* satisfies the high-efficiency requirement and the index-free requirement, Compared with *TPA*, *ResAcc* runs faster in the query phase by up to 6 times on all datasets. It is because *TPA* has to traverse the whole graphs by many iterations in the query phase. Compared with *BePI*, *ResAcc* answers SSRWR query faster even without the indexing structures on most datasets. It is because *BePI* needs to execute many matrix-vector multiplications, each of which requires $O(n^2)$ query time in the worst case. Moreover, *BePI* runs out of memory on large-scale datasets, e.g., Orkut and Twitter, which indicates that *BePI* is not scalable to large graphs. Compared with *FORA+*, *ResAcc* is slightly slower. However, *FORA+* suffers from the costly preprocessing time. For example, on Twitter, it takes *FORA+* around 1.5 hours to construct an index structure, which is unacceptable if graphs are changed dynamically. Moreover, *FORA+* runs out of memory in the preprocessing phase on large graphs (e.g., Friendster) since it needs to generate a huge number of random walks and consumes huge memory to store intermediate results. Besides, we evaluated the index updating time for each index-oriented approach when the graph is dynamically changed, the results indicate that without the large index updating time, *ResAcc* is a superior option for dynamic graphs than those index-oriented (see Appendix I for more details).

Besides, to verify the effect of each phase in *ResAcc*, we conducted an ablation study on *ResAcc*. Due to the space limit, the details and the results are shown in Appendix J. In summary, on average over 6 datasets, the h-HopFWD phase, the OMFWD phase and the remedy phase take about 1.79%, 64.58% and 33.63% of the total time, respectively. In addition, to demonstrate the effect of each trick used in *ResAcc* (the *accumulating loop* strategy, the h-hop induced subgraph, and the OMFWD phase), we compared *ResAcc* against different variants by removing each trick from *ResAcc*. For lack of space, the results are shown in Appendix K. In summary, all results demonstrate that each trick in *ResAcc* helps to improve the efficiency of *ResAcc*.

2) Accuracy: We proceed with the experiments measuring the accuracy of each approach (we only focus on approaches which guarantee relative errors as *ResAcc*) in terms of absolute error and NDCG. Firstly, following previous work [29], we reported the average absolute error of the k-th largest RWR values in Figure 4 where k is varied from $\{1, 10, 10^2, 10^3, 10^4, 10^5\}$. Due to space limit, the results on dataset WebStan could be found in Appendix A. Note that *BePI* on Orkut and Twitter are omitted since it runs out of memory. Besides, since *FORA+* has the same accuracy as *FORA*, we plotted the accuracy of *FORA* only. According to the results, the absolute error of *ResAcc* is among the smallest

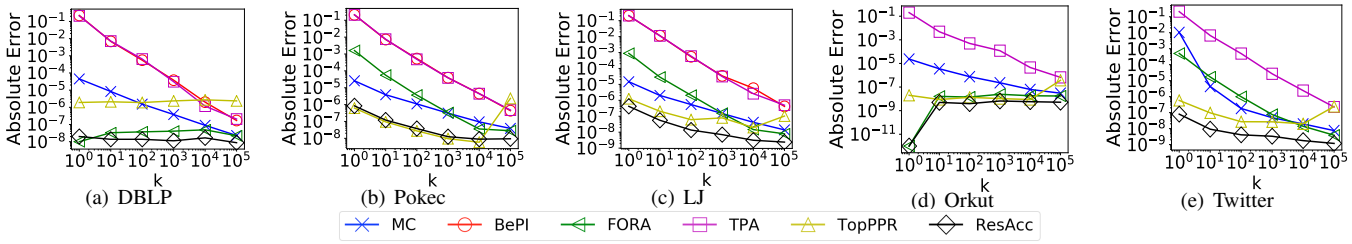


Fig. 4. The absolute error of each algorithm. BePI is omitted on Orkut and Twitter since it runs out of memory.

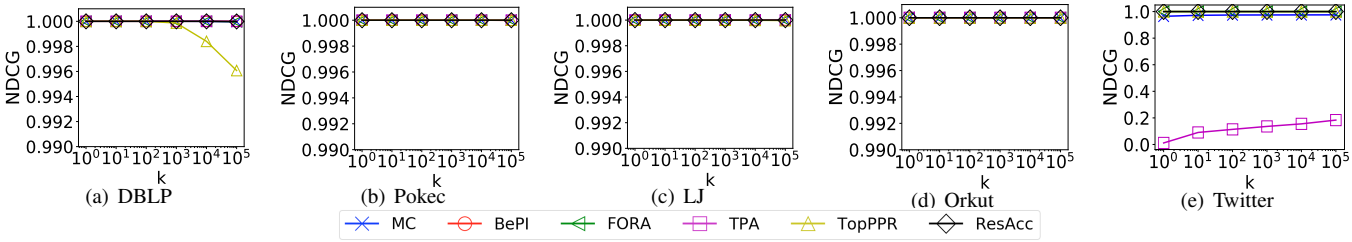
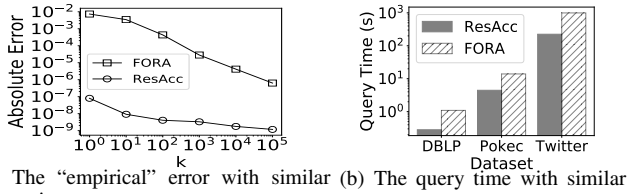
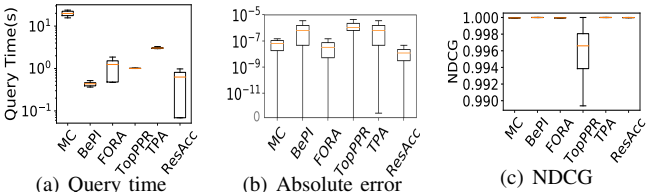


Fig. 5. The NDCG of each algorithm. BePI is omitted on Orkut and Twitter since it runs out of memory.



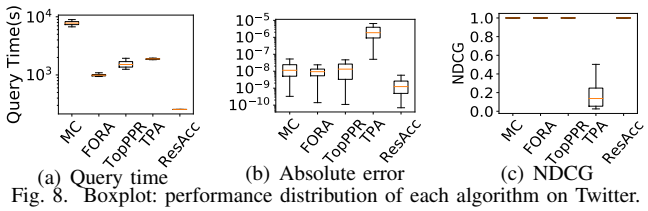
(a) The "empirical" error with similar query time (b) The query time with similar "empirical" error

Fig. 6. Fair comparison of *ResAcc* with *FORA*.



(a) Query time (b) Absolute error (c) NDCG

Fig. 7. Boxplot: performance distribution of each algorithm on DBLP.



(a) Query time (b) Absolute error (c) NDCG

Fig. 8. Boxplot: performance distribution of each algorithm on Twitter.

on all datasets. In particular, on Twitter, the absolute error of *ResAcc* is lower than that of *FORA* by up to 4 orders of magnitude. It is due to the h-hopFWD and OMFWD phases of *ResAcc* where a huge amount of residues are converted into reserves (a part of optimal RWR values) and so only small r_{sum} needs to be pushed further via the remedy process. However, *FORA* only converts a small amount of residues into reserves and estimates the RWR values by utilizing a lot of random walks (which are "randomized" RWR values).

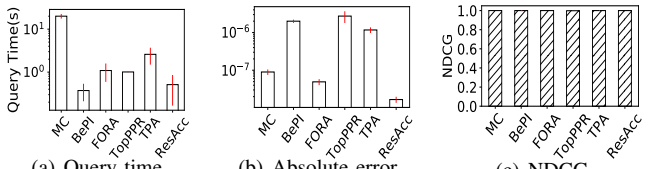
Secondly, in terms of NDCG (see Figure 5), we computed the NDCG value of each method by considering the k nodes with the highest RWR values returned by each method (where k is varied from $\{1, 10, 10^2, 10^3, 10^4, 10^5\}$). Our experiments show that all the methods except *TopPPR* and *TPA* can order the important nodes correctly on all dataset. Specifically, *TPA* has bad performance on Twitter (which is large-scale) since *TPA* approximates the RWR values for nodes which are not close to the source node by directly using their PageRank scores, which are not exactly the RWR values.

3) Fair comparison with FORA: For fair comparison with *FORA*, we evaluated two perspectives: (1) we measured the absolute error of results when *ResAcc* and *FORA* run in *similar* query times, and (2) we measured the query time when *ResAcc* and *FORA* output the results with *similar* absolute errors.

For the first perspective, we terminate the running of *FORA* as long as it takes as more query time than *ResAcc* in one specific dataset. We used Twitter for evaluation. The results in terms of absolute error are illustrated in Figure 6(a). We can see that *ResAcc* returns the values with much smaller absolute error than *FORA* by up to 6 orders of magnitude. It is because *FORA* cannot generate random walks from most of nodes in the graph when the time is over. For the second perspective, the details could be found in Appendix F for the lack of space. We evaluated on 3 datasets, namely DBLP, Pokec, and Twitter. The results are illustrated in Figure 6(b). We can see that *ResAcc* runs in less query time than *FORA* by up to around 4 times.

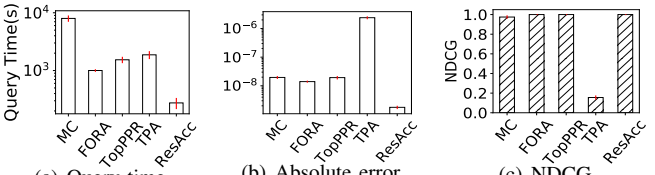
4) Performance for the outliers: In this section, we evaluated the performance distribution (instead of the average performance) of 6 methods, namely *MC*, *BePI*, *FORA*, *TopPPR*, *TPA* and *ResAcc* (we excluded other existing methods since they have been outperformed by these 6 methods in the previous section) on two datasets (i.e., DBLP and Twitter). Specifically, we use two visualization tools, namely "boxplot" (which reports *min*, *Q1*, *median*, *Q3*, and *max* among the results of all query nodes) and "error-bar" (which reports the mean and the standard deviation of all results), to show the performance distribution in terms of query time, absolute error and NDCG. The results plotted by "boxplot" are illustrated in Figure 7 and Figure 8, while the results plotted by "error-bar" are illustrated in Figure 9 and Figure 10. On dataset Twitter, the results of *BePI* are not plotted since it runs out of memory.

By "boxplot", the results show that *ResAcc* achieves better performance than other methods for handling the outliers in all



(a) Query time (b) Absolute error (c) NDCG

Fig. 9. Error-bar: performance distribution of each algorithm on DBLP.



(a) Query time (b) Absolute error (c) NDCG

Fig. 10. Error-Bar: performance distribution of each algorithm on Twitter.

terms of query time, absolute error and NDCG. Specifically, on Twitter, the maximum query time cost of *ResAcc* for a SSRWR query is the smallest. In addition, on Twitter, *ResAcc* has the lowest variability than all existing methods in terms of query time. Besides, *ResAcc* has the greatest accuracy among all methods in terms of absolute error. Similar findings could be found by using "error-bar". In summary, the results show that *ResAcc* achieves better performance than other methods for handling the outliers in 3 aspects.

C. Comparison with Particle Filtering

In this section, we examined the performance of *ResAcc* compared with *PF* in terms of average query time, absolute error and NDCG. We included *MC* for comparison since *PF* is a variant of *MC*. Besides, since *PF* has no accuracy guarantee, we set the total number of random walks used in *PF* to be equal to that in *MC* for fair comparison. We tested on DBLP and Twitter, and for each dataset, 50 source nodes were randomly selected. Besides, for each dataset, we set w_{min} to be 10^4 to optimize its performance in terms of efficiency and accuracy. Due to space limit, the results could be found in Appendix B. In summary, our experiments show that although *PF* takes similar query time to *ResAcc*, its performance in terms of absolute error and NDCG is outperformed by *ResAcc* by up to 4 orders of magnitude and nearly 3 times, respectively.

D. Effect of The Characteristics of Query Nodes

This section evaluated the performance of each method for the query nodes with the highest out-degrees. We included 4 index-free methods: *MC*, *FORA*, *TopPPR* and *ResAcc*, all of which have shown their superiority over other methods in previous sections. Specifically, we used two datasets, namely DBLP and Twitter, and chose 20 nodes with the largest out-degrees in each dataset. For the lack of space, the results are illustrated in Appendix C. In summary, our experiments show that *ResAcc* takes the least query time among all methods on all datasets. Besides, *ResAcc* achieves the highest accuracy than existing methods in terms of average absolute error.

E. Experimental Results for Multiple-Sources RWR Query

This section evaluates the performance of each algorithm for MSRWR query. We vary the number of sources $|\mathcal{S}|$ from $\{25, 50, 75, 100\}$, and used two datasets: DBLP and Twitter. We included two types of methods for comparison with *ResAcc*: the index-free methods (i.e., *MC*, *FORA* and *TopPPR*) and the index-based methods (i.e., *BePI*, *FORA+* and *TPA*). Besides, for each method, the average query time and the absolute error were evaluated (we excluded NDCG here since most of methods could order the nodes correctly, which have been shown in Section VII-B2). For the lack of space, the results could be found in Appendix D. In summary, the results show that *ResAcc* takes the least query time compared with the index-free methods by up to 2 orders of magnitude. Although *ResAcc* is slightly slower than *FORA+*, *ResAcc* avoids the heavy preprocessing cost and could be easily applied to large-scale dynamic graphs (while *FORA+* cannot), and *ResAcc* has higher accuracy than *FORA+*. Finally, *ResAcc* achieves the highest accuracy among all existing methods by up to nearly 3 orders of magnitude.

F. Fair Comparison with TopPPR

For fair comparison with *TopPPR*, we vary the value of $\mathcal{K}$ in *TopPPR* by setting it from from $\{5\times10^3, 1\times10^4, 5\times10^4, 1\times10^5, 5\times10^5\}$. For each $\mathcal{K}$, we evaluated the performance of *TopPPR* in terms of average query time, average absolute error, and NDCG of the k nodes with the highest RWR values on two datasets, namely DBLP and Twitter, where $k = 10^5$. Due to space limit, the results could be found in Appendix E. To sum up, our experiments show that *ResAcc* always takes less query time cost than *TopPPR* on both datasets by up to 2 orders of magnitude. Besides, with different k, *ResAcc* always achieves smaller absolute error than *TopPPR* by up to 2 orders of magnitude, and *ResAcc* always orders the important nodes correctly while *TopPPR* does not. Finally, we conducted an experiment to show the accuracy of both *ResAcc* and *TopPPR* when they take similar query time on Twitter. The results show that *ResAcc* achieves higher accuracy than *TopPPR* by up to 3 orders of magnitude.

G. Effect of r_{max}^{hop} in ResAcc

This section evaluated the effect of r_{max}^{hop} in *ResAcc*. The setting is as follows: we varied the value of r_{max}^{hop} from the set $\{10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}, 10^{-11}, 10^{-12}, 10^{-13}, 10^{-14}\}$ on DBLP. For parameters h and r_{max}^{f}, we set it by default (see Section VII-A). For each dataset, we measured the performance of *ResAcc* in terms of query time, absolute error and NDCG. Due to space limit, the experimental results could be found in Appendix H. In summary, *ResAcc* takes the least query time cost when r_{max}^{hop} is set to be 10^{-11}. Besides, the performance of *ResAcc* has non-monotonic behaviour with the value of r_{max}^{hop}. It is because a smaller value of r_{max}^{hop} makes the h-HopFWD phase take more query time to stop while a larger value makes the accumulated residues at the $(h+1)$-th layer smaller, leading to the OMFWD phase spend more query time. Thus, a proper choice of r_{max}^{hop} could optimize the performance of *ResAcc* in terms of query time.

H. ResAcc for Overlapping Community Detection

In this section, we examined the effectiveness of community detection using SSRWR queries and the effective-

TABLE V
THE EFFECT OF SSRWR QUERIES FOR COMMUNITY DETECTION.

Dataset	Method	Average Normalized Cut	Average Conductance
Facebook	*NISE* [30]	**0.2233**	**0.1917**
	NISE-without-SSRWR	0.5710	0.5601
DBLP	*NISE* [30]	**0.2365**	**0.2118**
	NISE-without-SSRWR	0.4719	0.4148

TABLE VI
THE RESULTS OF OVERLAPPING COMMUNITY DETECTION.

Dataset	Approach	Total Time (in seconds)	Average Normalized Cut	Average Conductance
Facebook	*FORA* [28]	3.8×10^3	0.2394	0.203
	ResAcc (ours)	$\mathbf{2.5 \times 10^3}$	0.2297	0.1950
DBLP	*FORA* [28]	1.5×10^4	0.2437	0.2151
	ResAcc (ours)	$\mathbf{6.4 \times 10^3}$	0.2373	0.2121

ness of *ResAcc* for overlapping community detection. Our experiments were conducted with *NISE* [30] (which adopts SSRWR queries as an important component for finding *high-quality* overlapping communities). Due to space limit, the experimental setting could be found in Appendix L. We used two common metrics in the literature to evaluate the quality of detected communities, namely *Average Normalized Cut* (ANC) and *Average Conductance* (AC). The smaller the value, the better the quality of communities. Table V and Table VI show the results of the effectiveness of community detection using SSRWR queries and the effectiveness of *ResAcc* for overlapping community detection, respectively. In summary, the results show that *ResAcc* is faster than *FORA* and returns the communities of better quality than *FORA*.

Summary: In summary, *ResAcc* outperforms most existing approaches in query time by up to 4 times, satisfying the high-efficiency requirement. Meanwhile, *ResAcc* not only guarantees the accuracy of the estimated RWR values (i.e., satisfies the output-bound requirement) but also has higher empirical accuracy than the state-of-the-art by up to 6 orders of magnitude. Finally, *ResAcc* is index-free and thus, it can be easily applied on both static and dynamic graphs. *ResAcc* is *the first algorithm* which satisfies all requirements for SSRWR simultaneously.

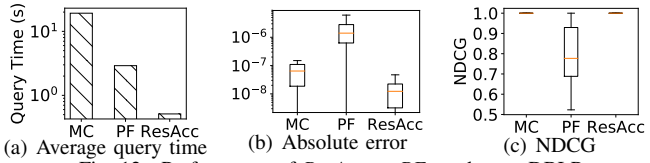
VIII. CONCLUSION AND FUTURE WORK

We present *ResAcc* for the approximate SSRWR query. *ResAcc* is based on the idea of *residue accumulation* so that it is able to avoid a mass of redundant computations, leading to higher efficiency than the existing algorithms. We provide the theoretical analysis of *ResAcc* in terms of both accuracy and query time. Extensive experiments demonstrate the superiority of *ResAcc* in terms of both efficiency and accuracy. Finally, the theoretic insight on why *ResAcc* is faster than *FORA* is an interesting future work due to its significant performance.

ACKNOWLEDGEMENT. We are grateful to the anonymous reviewers for their constructive comments on this paper. The research of Dandan Lin and Raymond Chi-Wing Wong was supported by HKRGC GRF 14205117.

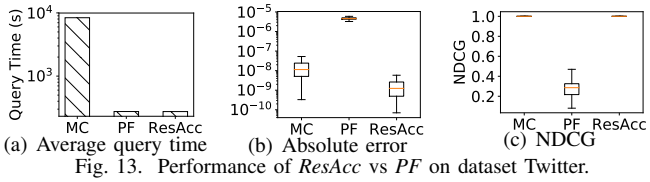
REFERENCES

[1] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. S Mirrokni, and S.-H. Teng. Local computation of pagerank contributions. In *International Workshop on Algorithms and Models for the Web-Graph*, 2007.
[2] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS'06*, 2006.
[3] L. Antonellis, H. G. Molina, and C. C. Chang. Simrank++: query rewriting through link analysis of the click graph. *VLDB*, 2008.
[4] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Journal on Numerical Analysis*, 2007.
[5] B. Cai, H. Wang, H. Zheng, and H. Wang. An improved random walk based clustering algorithm for community detection in complex networks. In *SMC*, 2011.
[6] S. Chakrabarti, A. Pathak, and M. Gupta. Index design and query processing for graph conductance search. *The VLDB Journal*, 2011.
[7] I. S. Dhillon, S. Mallela, and D. S Modha. Information-theoretic co-clustering. In *SIGKDD*, 2003.
[8] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *WWW'18*, 2018.
[9] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2005.
[10] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and etc. Fast and exact top-k search for random walk with restart. *VLDB*, 2012.
[11] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and etc. Efficient personalized pagerank with accuracy assurance. In *SIGKDD*, 2012.
[12] M. Girvan and M. EJ Newman. Community structure in social and biological networks. *NAS*, 2002.
[13] S. Godsill. Particle filtering: the first 25 years and beyond. In *ICASSP*. IEEE, 2019.
[14] J. Jung, N. Park, S. Lee, and U Kang. Bepi: Fast and memory-efficient method for billion-scale random walk with restart. In *SIGMOD*, 2017.
[15] N. Lao and W. W Cohen. Fast query execution for retrieval models based on path-constrained random walks. In *KDD*. ACM, 2010.
[16] W. Lin. Distributed algorithms for fully personalized pagerank on large graphs. In *WWW'19*, 2019.
[17] P. Lofgren, S. Banerjee, and A. Goel. Personalized pagerank estimation and search: A bidirectional approach. In *WSDM*, 2016.
[18] P. Nguyen, P. Tomeo, T. Di Noia, and E. Di Sciascio. An evaluation of simrank and personalized pagerank to build a recommender system for the web of data. In *WWW'15*, 2015.
[19] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.
[20] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu. Automatic multimedia cross-modal correlation discovery. In *SIGKDD*, 2004.
[21] W. J. Rugh. *Linear system theory*. 1996.
[22] K. Shin, J. Jung, S. Lee, and U Kang. Bear: Block elimination approach for random walk with restart on large graphs. In *SIGMOD*, 2015.
[23] H. Tong, C. Faloutsos, and J.Y. Pan. Fast random walk with restart and its applications. In *ICDM'06*, 2006.
[24] R. Wang, S. Wang, and X. Zhou. Parallelizing approximate single-source personalized pagerank queries on shared memory. *VLDB Journal*, 2019.
[25] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li. Hubppr: effective indexing for approximate personalized pagerank. *VLDB*, 2016.
[26] S. Wang and Y. Tao. Efficient algorithms for finding approximate heavy hitters in personalized pageranks. In *SIGMOD*, 2018.
[27] S. Wang, R. Yang, R. Wang, X. Xiao, and etc. Efficient algorithms for approximate single-source personalized pagerank queries. *TODS*, 2019.
[28] S. Wang, R. Yang, X. Xiao, Z. Wei, and etc. Fora: simple and effective approximate single-source personalized pagerank. In *SIGKDD*, 2017.
[29] Z. Wei, D. He, x. Xiao, S. Wang, S. Shang, and J.-R. Wen. Topppr: top-k personalized pagerank queries with precision guarantees on large graphs. In *SIGMOD*, 2018.
[30] J. J. Whang, D. F Gleich, and I. S Dhillon. Overlapping community detection using neighborhood-inflated seed expansion. *TKDE*, 2016.
[31] M. Yoon, J. Jung, and U Kang. Tpa: Fast, scalable, and accurate method for approximate random walk with restart on billion scale graphs. In *ICDE*. IEEE, 2018.



(a) Average query time (b) Absolute error (c) NDCG

Fig. 12. Performance of *ResAcc* vs *PF* on dataset DBLP.



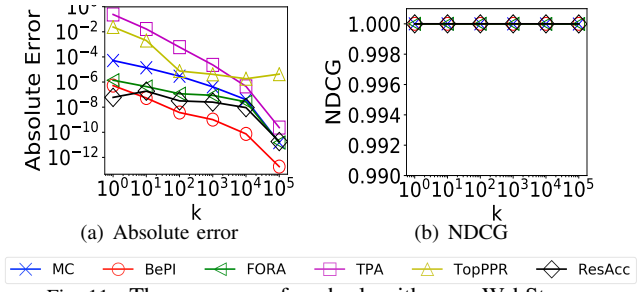
(a) Average query time (b) Absolute error (c) NDCG

Fig. 13. Performance of *ResAcc* vs *PF* on dataset Twitter.

APPENDIX

A. The accuracy results on WebStan

Figure 11 shows the absolute error of each algorithm on dataset Web-Stan in terms of absolute error. The results show that *ResAcc* has a smaller absolute error than *FORA* and *MC*. Although *BePI* has a lower error than other algorithms on Web-Stan, its error on other datasets are the highest (i.e., it has the lowest accuracy), indicating that *BePI* cannot provide stable accuracy guarantee. It is because *BePI* cannot bound the relative error between the estimated and optimal RWR value of each node. In comparison, *ResAcc* not only guarantees a relative error theoretically (and thus, it satisfies the output-bound requirement), but also returns results with high and stable accuracy empirically.



(a) Absolute error (b) NDCG

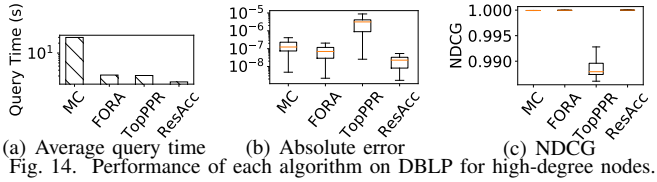
Fig. 11. The accuracy of each algorithm on WebStan.

B. Comparison with Particle Filtering

The results are illustrated in Figure 12 and Figure 13. Our experiments show that *ResAcc* outperforms *PF* in terms of both efficiency and accuracy. Although *PF* takes similar query time to *ResAcc*, its performance in terms of absolute error and NDCG is outperformed by *ResAcc* by up to 3 orders of magnitude and nearly 3 times, respectively. It is because the "randomized" process of *PF* only selects the out-neighbours of each node which has non-zero random walks after the "deterministic" distribution process, which is of flaws since it constrains the lengths of each random walks.

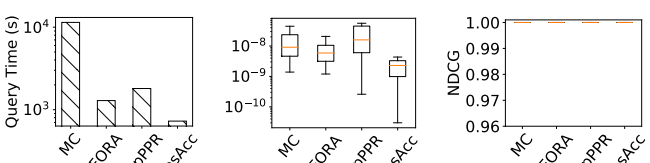
C. Effect of The Characteristics of Query Nodes

The results are illustrated in Figure 14 and Figure 15. Our experiments show that *ResAcc* has the best performance among all methods on both datasets. Specifically, *ResAcc* takes the least query time among all methods. Besides, *ResAcc* achieves



(a) Average query time (b) Absolute error (c) NDCG

Fig. 14. Performance of each algorithm on DBLP for high-degree nodes.



(a) Average query time (b) Absolute error (c) NDCG

Fig. 15. Performance of each algorithm on Twitter for high-degree nodes.

the highest accuracy than existing methods in terms of average absolute error (i.e., the line in orange color). Thus, *ResAcc* is more robust for handling the "hub" nodes with high out-degree.

D. Experimental Results for Multiple-Sources RWR Query

The results are shown in Figure 16 and Figure 17. On both datasets, the results demonstrate that the query time of each method increases as the number of source nodes increases, while the absolute error of each method does not change significantly as the number of source nodes increases. Besides, our experiments show that *ResAcc* takes the least query time compared with the index-free methods by up to 2 orders of magnitude. Although *ResAcc* is slightly slower than *FORA+*, *ResAcc* avoids the heavy preprocessing cost and could be easily applied to large-scale dynamic graphs (while *FORA+* cannot), and *ResAcc* has higher accuracy than *FORA+*. Finally, *ResAcc* achieves the highest accuracy among all existing methods in terms of absolute error by up to nearly 3 orders of magnitude. Thus, *ResAcc* outperforms all existing methods in terms of both efficiency and accuracy, which is consistent with the results for SSRWR query.

E. Fair comparison with TopPPR

For fair comparison with *TopPPR*, we vary the value of $\mathcal{K}$ in *TopPPR* by setting it from from $\{5\times10^3, 1\times10^4, 5\times10^4, 1\times10^5, 5\times10^5\}$. For each $\mathcal{K}$, we evaluated the performance of *TopPPR* in terms of query time, absolute error, and NDCG on two datasets, namely DBLP and Twitter. The results show that with different $\mathcal{K}$, *ResAcc* always takes less query time cost than *TopPPR* on both datasets by up to 2 orders of magnitude. It is because *TopPPR* needs to performs the backward search from each node which are possible in the top-$\mathcal{K}$ set in each iteration while *ResAcc* not. Besides, with different $\mathcal{K}$ *ResAcc* always achieves higher absolute error than *TopPPR* by up to 2 orders of magnitude. In addition, with different $\mathcal{K}$ *ResAcc* always orders the k nodes (where $k = 10^5$) with the highest RWR values correctly while *TopPPR* not. It is because *TopPPR* terminates once it is "confident" with probability that the top-$\mathcal{K}$ nodes have been found, which, however, cannot estimate the RWR values of each node in the graph with accuracy guarantee. Finally, *TopPPR* is not suitable for SSRWR query

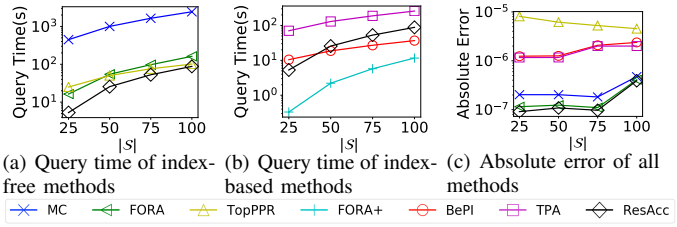


Fig. 16. Performance of each method for MSRWR query on DBLP.

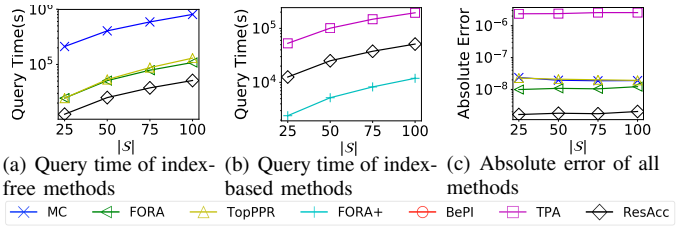


Fig. 17. Performance of each method for MSRWR query on Twitter.

since its performance varied significantly when varying k on different datasets. Specifically, the query time of *TopPPR* on DBLP is the least when $\mathcal{K} = 5 \times 10^4$, while its query time on Twitter increases as $\mathcal{K}$ increases. It is because the query time cost of *TopPPR* depends on the gap between the $\mathcal{K}$-th and the $(\mathcal{K}+1)$-th largest RWR values, which varied largely on different datasets.

Besides, for fair comparsion, we conducted an experiment by letting *TopPPR* runs in the similar query time to *ResAcc*, and measured their accuracy of the results in terms of absolute error and NDCG. Here, we teseted on Twitter. In particular, we first set the parameter $\mathcal{K}$ used in *TopPPR* to be 3000, and terminates the program if the query time of *TopPPR* exceeds that of *ResAcc* (i.e., 275 seconds). In addition, for absolute error, we plot the absolute error of the k-th largest RWR value, and for NDCG, we plot the ndcg for the k nodes with highest RWR values, where k is varied from $\{1, 10, 10^2, 10^3, 10^4, 10^5\}$. The results are shown in Figure 20. Our experiments show that *ResAcc* achieves higher accuracy than *TopPPR* in terms of absolute error by up to 3 orders of magnitude, while running in similar query time. Figure 20(b), *TopPPR* cannot corretly orders the nodes whose RWR values are the k-th highest where $k = 10^4$ and $k = 10^5$. It is because *TopPPR* performs the backward searches only from the nodes which are possiblly in the top-$\mathcal{K}$ set, instead of all nodes in the graph. This reason can explain why the absolute error of *TopPPR* will decrease as k increase and then increase as k increase (see Figure 20(a)).

F. Fair comparison with FORA

This section shows the experimental setting for the second perspective of fair comparison with *FORA* mentioned in Section VII-B3. For the second perspective, we controlled the "empircal" error of *ResAcc* by adjusting the number of random walk used in the remedy phase for easy implementation. We adjusted the number of random walks by $n_{scale} \cdot n_r$ where

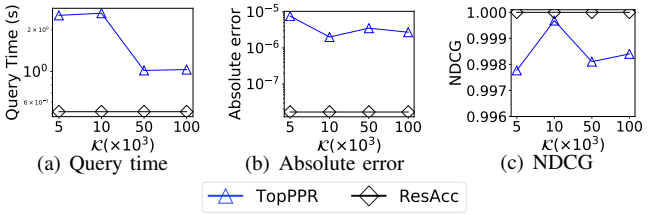


Fig. 18. Fair comparison of *ResAcc* with *TopPPR* on DBLP.

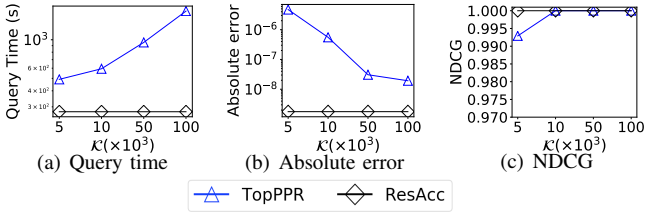


Fig. 19. Fair comparison of *ResAcc* with *TopPPR* on Twitter.

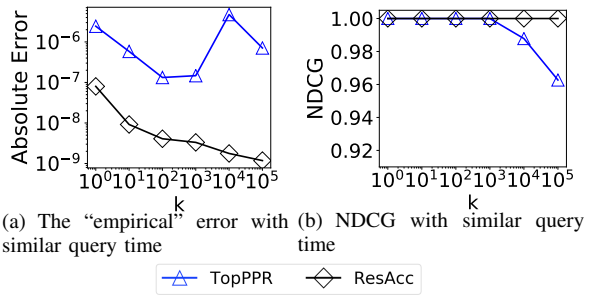
n_{scale} is a parameter whose values are chosen from {0, 0.2,0.4,0.6,0.8,1.0}. The algorithm terminates as long as its average "empirical" error satisfies the following requirement: $|err_{res} - err_f| < 0.1 \cdot err_f$ where err_{res} and err_f are the average absolute errors of RWR values of all nodes estimated by *ResAcc* and *FORA*, respectively. We evaluated on 3 datasets, namely DBLP, Pokec, and Twitter.

G. Effect of parameter h

In this section, we evaluated the effect of parameter h in *ResAcc* in terms of query time. We evaluated on two datasets, namely Web-Stan (a small dataset) and Pokec (a large dataset). Besides, the value of h is varied from $\{1, 2, 3, 4, 5, 6\}$. For fair comparison, we also included the time cost of *FORA*. The results are illustrated in Figure 21. From the results, we can see that when h is set to be a smaller value (e.g., $1 \leq h \leq 4$), *ResAcc* runs faster than *FORA* by at least 2 times on both datasets. Moreover, in both datasets, *ResAcc* takes the least query time when $h = 2$. Specifically, when $h > 2$, the query time of *ResAcc* increases as h increases. It is because that the h-hop set of h-HopFWD will include more node when h increases, and thus, *ResAcc* takes more time to finish the h-HopFWD phase. On the other hand, when $h < 2$, the query time of *ResAcc* decreases as h increases. In summary, a small value of h is suitable for both small and large datasets. A simple solution for choosing h for a dataset is to directly set h to be 2 according to the above experiments. In our experiments on other datasets, we sets h be 2 for most datasets, and the results shows that *ResAcc* runs faster than *FORA* by around 4 times (see Section VII-B1).

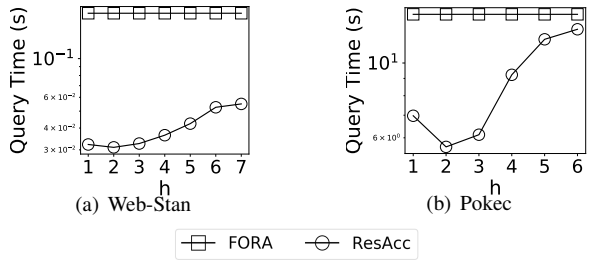
H. Effect of r_{max}^{hop} in ResAcc

This section elaborates the experimental results of r_{max}^{hop} in *ResAcc*, which are shown in Figure 22. The results show that *ResAcc* takes the least query time when r_{max}^{hop} is 10^{-11}. Specifically, when $r_{max}^{hop} < 10^{-11}$, the query time of *ResAcc* increases when r_{max}^{hop} decreases. It is because the smaller value of r_{max}^{hop} in this case makes the h-HopFWD phase take more time to stop the forward searching phase. When



(a) The "empirical" error with similar query time (b) NDCG with similar query time

Fig. 20. The accuracy of *ResAcc* vs *TopPPR* with similar query time on Twitter.



(a) Web-Stan (b) Pokec

Fig. 21. The effect of h in *ResAcc*.

$r_{max}^{hop} > 10^{-11}$, the query time of *ResAcc* decreases when r_{max}^{hop} decreases. It is because, in this case, a smaller value of r_{max}^{hop} makes the residues of nodes in $L_{(h+1)-hop}(s)$ be accumulated to a larger value, and so helps *ResAcc* run faster in the *OMFWD* phase. Moreover, *ResAcc* achieves the highest accuracy in terms of absolute error when r_{max}^{hop} is equal to 10^{-14} . Besides, with different r_{max}^{hop}, *ResAcc* alwasys orders the important nodes correctly.

I. Dynamic updating cost

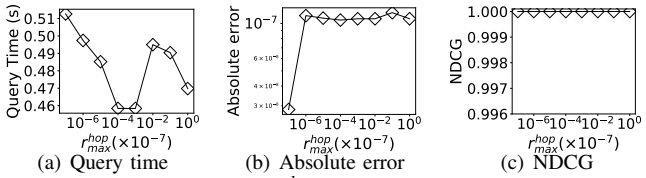
Here, we evaluated the index updating time for each index-oriented approach when the graph is dynamically changed and the results are presented in Figure 23. Specifically, we randomly deleted 50 nodes on each dataset and reported the average index updating time of each approach for each node deletion. Recall that *ResAcc* is an index-free approach. Its updating time is **zero**, while *BePI* and *FORA+* have to re-build the index from scratch for each deletion, which is very expensive. In particular, *BePI* runs of memory on Orkut and Twitter in the preprocessing phase, showing its limited applicability on large-scale graphs. Considering the large index updating time, *ResAcc* is a superior option for processing dynamic graphs than those index-oriented.

J. Breakdown Time Cost of ResAcc

In this section, we give the query time cost of each phase in *ResAcc*. The results in each dataset are shown in Table VII.

K. Effect of each Trick used in ResAcc

In this section, we aim to show the effect of tricks used in *ResAcc*, namely the accumulating loop strategy in the h-HopFWD phase, the h-hop induced subgraph in the h-HopFWD phase, and the OMFWD phase. The results are illustrated in Figure 24.



(a) Query time (b) Absolute error (c) NDCG

Fig. 22. The effect of r_{max}^{hop} of *ResAcc* on dataset DBLP.

TABLE VII
THE QUERY TIME (IN SECONDS) OF EACH PHASE IN *ResAcc* VS. DATASET.

	DBLP	Web-Stan	Pokec	LJ	Orkut	Twitter
h-HopFWD	0.0144	0.0006	0.015	0.022	0.5662	8.4696
OMFWD	0.302	0.0208	3.6828	7.601	15.3128	181.861
Remedy	0.1962	0.0096	1.9406	4.3316	7.185	84.3918
Total	0.5126	0.031	5.6384	11.9546	23.064	274.722

Effect of the Accumulating Loop strategy. We remove the accumulating loop strategy in the h-HopFWD phase and used the general forward search algorithm in the h-hop induced subgraph. We denote the new approach as *No-Loop-ResAcc*. All parameters set for *No-Loop-ResAcc* are the same as *ResAcc*. Figure 24(a) shows the query time of *No-Loop-ResAcc* and *ResAcc* in each dataset. We can see that *ResAcc* runs faster than No-Loop-ReAcc by at least 2 times. It is because *ResAcc* used the *accumulating loop* strategy at the source node to reduce the time cost in the h-HopFWD phase.

Effect of the h-hop induced subgraph. To test the effect of the h-hop induced subgraph, in the h-HopFWD phase, we uses the accumulating loop strategy in the whole graph instead of only the h-hop subgraph. We denote the new approach as *No-SG-ResAcc*. All parameters set for *No-SG-ResAcc* are the same as *ResAcc*. Figure 24(b) shows the query time of *No-SG-ResAcc* and *ResAcc* in each dataset. We can see that *ResAcc* runs faster than No-SG-ResAcc by up to 2 times. It is because *ResAcc* can quickly perform the push operations in the subgraph instead of the whole graph. Besides, the h-hop subgraph in the *ResAcc* can accumulated the large value for the nodes in the $(h+1)$-th hop layer, beneficial for the high efficiency in the OMFWD phase.

Effect of the OMFWD phase. In this section, we showed the effect of the OMFWD phase in *ResAcc*. We remove the OMFWD phase in *ResAcc* so that it only contains the h-hopFWD phase and the remedy phase. We denote the new approach as *No-OFD-ResAcc*. From the results, we can see that *ResAcc* runs faster than *No-OFD-ResAcc* by up to 1 order of magnitude. It is because the OMFWD phase in *ResAcc* is able to further reduce the sum of residues of the nodes in the graph, leading to less number of random walks to be simulated.

L. ResAcc for overlapping community detection

In this section, we first give the definition of overlapping community detection problem, and then give the definitions of two metrics used for evaluating the quality of discovered communities. Finally, we show the experimental results for

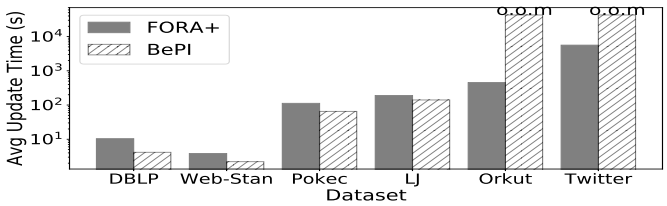


Fig. 23. The Index updating time of index-oriented approaches for per node deletion in the dynamic graph. The word "o.o.m" stands for "out of memory".

showing the superiority of SSRWR queries in *NISE*.

Problem definition [30]. The goal of the traditional community detection is to partition a graph $G(V, E)$ into $|C|$ pairwise disjoint communities: $\mathcal{C}_1, ..., \mathcal{C}_{|C|}$ such that $\mathcal{C}_1 \cup ... \cup \mathcal{C}_{|C|} = V$. On the other hand, the goal of the overlapping community detection problem is to find overlapping communities whose union is not necessarily equal to the entire nodes set V. Formally, it seeks $|C|$ overlapping communities such that $\mathcal{C}_1 \cup ... \cup \mathcal{C}_{|C|} \subseteq V$.

Average Normalized Cut [30]. Let $links(\mathcal{C}_i, \mathcal{C}_j)$ denote the sum of edges linking between two communities $\mathcal{C}_i$ and $\mathcal{C}_j$. Let $Cut(\mathcal{C}_i)$ denote the cut of a community $\mathcal{C}_i$, which is the sum of edges linking between $\mathcal{C}_i$ and its complement $V - \mathcal{C}_i$. Then, the normalized cut of a community $\mathcal{C}_i$ is defined by the cut with normalization as follows:

$$ncut(\mathcal{C}_i) = \frac{Cut(\mathcal{C}_i)}{links(\mathcal{C}_i, V)}.$$

Finally, the average normalized cut is defined to be the average normalized cut of all detected communities such that:

$$avg_{ncut} = \frac{1}{|C|} \sum_{i=1}^{|C|} ncut(\mathcal{C}_i).$$

Average Conductance [30]. The conductance of a community is defined to be the cut divided by the least number of edges incident on eight set $\mathcal{C}_i$ or $V - \mathcal{C}_i$:

$$cond(\mathcal{C}_i) = \frac{cut(\mathcal{C}_i)}{\min(links(\mathcal{C}_i, V), links(V - \mathcal{C}_i, V)}.$$

Then, the average conductance is defined by the average conductance of all detected communities such that:

$$avg_{cond} = \frac{1}{|C|} \sum_{i=1}^{|C|} cond(\mathcal{C}_i).$$

The effectiveness of SSRWR queries in *NISE*. In order to show the effectiveness of SSRWR queries in *NISE*, we conducted an experiment by comparing the communities returned by the original version of *NISE* (which adopts SSRWR) with those by the modified version of *NISE* without using SSRWR. Specifically, *NISE* (with SSRWR) processes nodes in descending order of RWR values (from each of the "chosen" nodes), while *NISE* (without SSRWR) in descending order of distances. With the difference of ordering being processed, the communities found are also different. We used two common metrics in the literature to evaluate the quality of detected communities, namely *Average Normalized Cut* (ANC) and *Average Conductance* (AC). For both metrics, the smaller the value, the better the quality of communities. Table V shows the results on two datasets, namely Facebook and DBLP. We can see that (the original version of) *NISE* returns communities with better quality compared with *NISE without SSRWR* by up to 2.5 times and by up to 2.9 times in terms of *ANC* and *AC*, respectively. Thus, SSRWR is effective for the community detection.

***ResAcc* for overlapping community detection.**

For examining the effectiveness of *ResAcc* for overlapping community detection, the followings show the detailed setting. We tested on two datasets following [30], namely (1) *Facebook* with 4,039 nodes and 176,470 edges and (2) *DBLP* with 0.3 million nodes and 2.1 million edges. Let $|C|$ denote the number of communities to be found in a graph. Following [30], $|C|$ is set to $10,000$ and 200 for Facebook and DBLP, respectively. We used either *FORA* or *ResAcc* as an algorithm for each SSRWR query in *NISE*. We reported the total time cost for each dataset for evaluating the efficiency. Then, following [30], two metrics for evaluating the quality of results are used: *Average Normalized Cut* and *Average Conductance*. Table VI shows the results of community detection. The results show that *ResAcc* is faster than *FORA* by up to 2 times. Moreover, *ResAcc* returns the communities of better quality than *FORA*. In particular, the communities detected by *ResAcc* has lower average normalized cut and lower average conductance than those by *FORA* on both datasets by up to 4.05% and by up to 3.94%, respectively.

M. Proof of Theorem 1

Proof. From Algorithm 2, we have $E[\hat{\pi}(s,t)] = \pi^f(s,t) + E[C_t]$. To prove that $E[\hat{\pi}(s,t)] = \pi(s,t) = \pi^f(s,t) + \sum_{v \in V} r^f(s,v) \cdot \pi(v,t)$, it is equal to prove that $E[C_t] = \sum_{v \in V} r^f(s,v) \cdot \pi(v,t)$. Let v be the node whose residue is larger than zero. Now, we first consider only the $n_r(v)$ random walks that starts from v. Let $X_i(t)$ be a Bernoulli variable that

$$X_i(t) = \begin{cases} 1, & \text{if the } i\text{-th walk that starts from } v \text{ ends at } t, \\ 0, & \text{otherwise.} \end{cases}$$

By definition, we have $E[X_i(t)] = \pi(v,t)$. The amount of increment that C_t receives when *ResAcc* processes node v (see Lines 10-15 in Algorithm 2) is $\sum_{i=1}^{n_r(v)} (\frac{a(v) \cdot r_{sum}}{n_r} \cdot X_i(t))$. Then, based on the definition of $n_r, n_r(v)$, and $a(v)$, we have

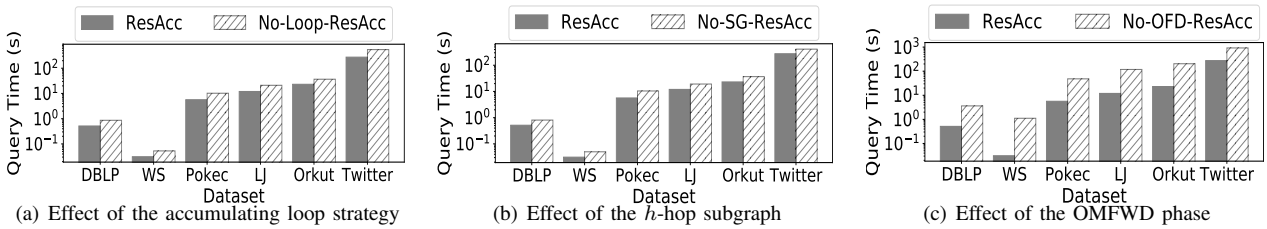
$$E[\sum_{i=1}^{n_r(v)} (\frac{a(v) \cdot r_{sum}}{n_r} \cdot X_i(t))] = r^f(s,v) \cdot \pi(v,t). \quad (6)$$

By considering all such node v whose residue is non-zero, we can obtain that $E[C_t] = \sum_{v \in V} r^f(s,v) \cdot \pi(v,t)$. So, the proof completes. □

N. Proof of Lemma 1

Proof. To apply Theorem 2, let us consider the $n_r = \sum_{v \in V} n_r(v)$ random walks simulated by *ResAcc* where $n_r(v)$ is the number of random walks that starting from node v as defined in our paper. Let $b_j = a(v)$ if the j-random walk starts from a node $v \in V$ where $j \in \{1, ..., n_r\}$. Then, we can know that $max_j b_j = 1$, and $b_j^2 \leq b_j$ for any j. In addition, we define $Y_j(t)$ be the random variable such that:

$$Y_j(t) = \begin{cases} 1, & \text{if the } j\text{-th walk ends at } t, \\ 0, & \text{otherwise.} \end{cases}$$



(a) Effect of the accumulating loop strategy (b) Effect of the h-hop subgraph (c) Effect of the OMFWD phase

Fig. 24. Effect of each trick used in *ResAcc*. The dataset "WS" is short for "Web-Stan".

Next, based on these definitions, we give the proof of Lemma 1. Firstly, we define $Y = \frac{1}{n_r}\sum_{j=1}^{n_r} b_j Y_j(t)$, and $v = \frac{1}{n_r}\sum_{j=1}^{n_r} b_i^2 \cdot E[Y_j(t)]$. Let $a = \max\{b_1, ..., b_{n_r}\}$. By definition, $b_j^2 \leq 1$, and so, $v \leq E[Y]$ and $a \leq 1$. By using Theorem 2, for any λ, we have that

$$Pr[|Y - E[Y]| \geq \lambda] \leq 2 \cdot \exp(-\frac{\lambda^2 \cdot n_r}{2v + 2a\lambda/3}).$$

By applying $v \leq E[Y]$ to above inequality, we have that:

$$Pr[|Y - E[Y]| \geq \lambda] \leq 2 \cdot \exp(-\frac{\lambda^2 \cdot n_r}{2E[Y] + 2a\lambda/3}).$$

Besides, from our definition, we can observe that

$$\pi(s,t) - \hat{\pi}(s,t) = \frac{n_r(v) \cdot r_{sum}}{n_r}(E[Y] - Y),$$

because of Equation 6. Thus, the above inequality could be rewritten as:

$$Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \frac{n_r(v) \cdot r_{sum}}{n_r} \cdot \lambda]$$

$$\leq 2\exp(-\frac{\lambda^2 \cdot n_r}{2E[Y] + 2a\lambda/3}).$$

Besides, we know that $E[Y] \leq \frac{n_r}{n_r(v) \cdot r_{sum}} \cdot \pi(s,t)$ from Equation 6. By replacing $E[Y]$, the following equality could be obtained:

$$Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \frac{n_r(v) \cdot r_{sum}}{n_r}\lambda]$$

$$\leq 2\exp(-\frac{\lambda^2 \cdot n_r}{2\frac{n_r \cdot \pi(s,t)}{n_r(v) \cdot r_{sum}} + 2a\lambda/3}).$$

Let $\lambda = \epsilon \cdot \frac{n_r \cdot \pi(s,t)}{n_r(v) \cdot r_{sum}}$, we have:

$$Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon \cdot \pi(s,t)] \leq 2\exp(-\frac{\epsilon^2 \cdot n_r \cdot \pi(s,t)}{r_{sum} \cdot (2 + 2a\epsilon/3)}).$$

Since $a \leq 1$, we get that:

$$Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon \cdot \pi(s,t)] \leq 2\exp(-\frac{\epsilon^2 \cdot n_r \cdot \pi(s,t)}{r_{sum} \cdot (2 + 2\epsilon/3)}).$$

Now, the proof completes. □

O. Proof of Theorem 3

Proof. Since $n_r = r_{sum} \cdot \frac{(2\epsilon/3+2) \cdot \log(2/p_f)}{\epsilon^2 \cdot \delta}$, according to Lemma 1, we have that

$$Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon\pi(s,t)] \leq 2\exp(-\frac{\epsilon^2 \cdot n_r \cdot \pi(s,t)}{r_{sum} \cdot (2 + 2\epsilon/3)})$$

$$= 2\exp(-\frac{\epsilon^2 \cdot \pi(s,t)}{r_{sum} \cdot (2 + 2\epsilon/3)} \cdot r_{sum} \cdot \frac{(2\epsilon/3 + 2) \cdot \log(2/p_f)}{\epsilon^2 \cdot \delta})$$

Since $\pi(s,t) > \delta$, we have that

$$Pr[|\pi(s,t) - \hat{\pi}(s,t)| \geq \epsilon \cdot \pi(s,t)] \leq 2\exp(\log(2/p_f)) = p_f.$$

So, the proof completes. □

P. Proof of Lemma 2

Proof. For the case with $r_0^f(s,s) = 1$, we assume that the total number of push operations is l. We denote the ordering of nodes selected for the push operations as $\{N\}_l = \{v_1, v_2, ..., v_l\}$ where $v_i \in V$. Similarly, We denote the ordering of nodes selected with $r_1^f(s,s)$ as $\{N'\}_{l'} = \{v_1^{'}, v_2^{'}, ..., v_{l'}^{'}\}$ where l' is the number of push operations done with $r_1^f(s,s)$. In the following, we prove that $\{N\}$ is identical to $\{N'\}$ such that: (i) $l = l'$ and (ii) $v_i = v_i^{'}$ for each $v_i \in \{N\}_l$.

According to the accumulating phase, we know that the first node in $\{N\}_l$ must be the source node, i.e., $v_1 = s$ whose residue $r_0^f(s,s) \geq r_{max}^f \cdot d_{out}(s)$. By multiplying $r_1^f(s,s)$ at each side of the above inequality, we have $r_1^f(s,s) \geq r_{max}^f \cdot r_1^f(s,s) \cdot d_{out}(s)$. Thus, the source node s also satisfies the push condition for the case with $r_1^f(s,s)$, and is the first node in $\{N'\}$.

Assume that $v_j = v_j^{'}$ for each $j \in [1, i-1]$. We aim to prove that $v_i = v_i^{'}$. Firstly, the following inequality holds: $r_0^f(s,v_i) \geq r_{max}^f \cdot d_{out}(v_i)$. Besides, following the process of the accumulating phase, we have the following invariant for node v_i. $r_0^f(s,v_i) = c_i \cdot r_0^f(s,s)$ where $c_i \in (0,1)$ is the probability of all possible pathes from s to v_i that go through v_j. This equation indicates that the residue of any node is proportional to the initial residue $r_0^f(s,s)$. Similarly, for node $v_i^{'}$, we have $r_1^f(s,v_i^{'}) = c_i^{'} \cdot r_1^f(s,s)$. Since $v_j = v_j^{'}$ for each $j \in [1, i-1]$, we have $c_i = c_i^{'}$. Thus, we have $r_1^f(s,v_i^{'}) = \frac{r_0^f(s,v_i)}{r_0^f(s,s)} \cdot r_1^f(s,s) \geq r_{max}^f \cdot d_{out}(v_i) \cdot r_1^f(s,s)$ which satisfies the push condition. In other words, $v_i = v_i'$. Hence, $\{N\}$ is identical to $\{N'\}$. The proof completes. □

Q. Proof of Lemma 3

Proof. For simplicity, we introduce a straightforward method, which is slightly different from h-HopFWD. We denote this method as *One-Accumulating-One-Pushing* (*OAOP*). After the first push operation with $r_0^f(s,s)$, *OAOP* accumulates the residue of s ($r_1^f(s,s)$) until it value cannot be changed any more, and then pushes the accumulated residue $r_1^f(s,s)$ at source node again, which triggers the accumulation in the next iteration. *OAOP* terminates until the residue $r_T^f(s,s)$ cannot satisfy the push condition where T is the T-th iteration. It means T should satisfy the following two conditions: (1) the residue of s obtained by the T-th accumulating phase do not satify the *pushing condition*, i.e., $r_T^f(s,s)/d_{out}(s) > r_{max}^{hop}$; (2) the residue of s obtained by the $(T-1)$-th

accumulating phase must satify the *pushing condition*, i.e., $r^f_{T-1}(s,s)/d_{out}(s) \leq r^{hop}_{max}$. By solving these two inequalities, we have $\frac{\log[r^{hop}_{max} \cdot d_{out}(s)]}{\log r^f_1(s,s)} < T \leq \frac{\log[r^{hop}_{max} \cdot d_{out}(s)]}{\log r^f_1(s,s)} + 1$.

Let $r^f_i(s,s)$ denote the residue of s obtained in the i-th iteration of *OAOP*. By adjusting the push condition according to Lemma 2, when *OAOP* pushes $r^f_i(s,s)$ (which starts the $(i+1)$-th iteration), we have $\pi^f_{i+1}(s,t) = \pi^f_1(s,t) \times r^f_i(s,s)$ which holds for any node $t \in V$, where $\pi^f_{i+1}(s,t)$ is the part of reserve of node t obtained in the $(i+1)$-th iteration only. By summing up the part of reserve that node t receives in all iterations, we have

$$\pi^f(s,t) = \pi^f_1(s,t) + \sum_{i=2}^{T} \pi^f_i(s,t) = \sum_{i=1}^{T} \pi^f_1(s,t) \times r^f_{i-1}(s,s).$$

Since $r^f_i(s,s) = [r^f_1(s,s)]^{i-1}$ where $i \geq 2$, we have

$$\pi^f(s,t) = \sum_{i=1}^{T} \pi^f_1(s,t) \times [r^f_1(s,s)]^{i-1} = \pi^f_1(s,t) \times S$$

where $S = \sum_{i=1}^{T} [r^f_1(s,s)]^{i-1} = \frac{1-[r^f_1(s,s)]^{T-1}}{1-r^f_1(s,s)}$. Similarly, we can get that $r^f(s,t) = r^f_1(s,t) \times S$. Thus, the proof completes. □

R. Proof of Lemma 4

Proof. From the definition of RWR, we have the invariant that $r^{hop}_{sum} + \sum_{v \in V_{h-hop}(s)} \pi^f(s,v) = 1$ after h-HopFWD terminates. This invariant means that r^{hop}_{sum} is larger if the sum of the reserves of nodes in $V_{h-hop}(s)$ is smaller, and vice versa. As we know, the residue of each node v in $V_{h-hop}(s)$ is reduced if there exists the forward push operations at v during the process of h-HopFWD. Thus, the conclusion can be drawn that r^{hop}_{sum} is largest if h-HopFWD does only a single push operation at each $v \in V_{h-hop}(s)$, which happens when the structure of the h-hop induced subgraph is as follows: for any two nodes $v_1 \in L_{i-hop}(s)$ and $v_2 \in L_{j-hop}(s)$ where $0 \leq i < j \leq h$, there does not exist an edge linking from v_2 to v_1. Based on the above structure, we have $r^{hop}_{sum} = 1 - \sum_{v \in V_{k-hop}(s)} \pi^f(s,v) = 1 - \sum_{j=0}^{h} \sum_{v \in L_{j-hop}(s)} \pi^f(s,v)$. Since for each layer j such that $0 \leq j \leq h$, $\sum_{v \in L_{j-hop}(s)} \pi^f(s,v) = \alpha \cdot \sum_{v \in L_{j-hop}(s)} r^f(s,v) = \alpha \cdot \sum_{u \in L_{(j-1)-hop}(s)} \frac{1-\alpha}{d_{out}(u)} \cdot r^f(s,u) = \alpha \cdot (1-\alpha)^j$, we have $r^{hop}_{sum} = 1 - \alpha \cdot [1 + (1-\alpha) + ... + (1-\alpha)^h] = (1-\alpha)^h$. So the proof completes. □