Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme

GUANHAO HOU, The Chinese University of Hong Kong, Hong Kong SAR
QINTIAN GUO, The Chinese University of Hong Kong, Hong Kong SAR
FANGYUAN ZHANG, The Chinese University of Hong Kong, Hong Kong SAR
SIBO WANG*, The Chinese University of Hong Kong, Hong Kong SAR
ZHEWEI WEI, Renmin University of China, China

Personalized PageRank (PPR) stands as a fundamental proximity measure in graph mining. Given an input graph G with the probability of decay α, a source node s and a target node t, the PPR score $\pi(s,t)$ of target t with respect to source s is the probability that an α-decay random walk starting from s stops at t. A *single-source PPR (SSPPR)* query takes an input graph G with decay probability α and a source s, and then returns the PPR $\pi(s,v)$ for each node $v \in V$. Since computing an exact SSPPR query answer is prohibitive, most existing solutions turn to approximate queries with guarantees. The state-of-the-art solutions for approximate SSPPR queries are index-based and mainly focus on static graphs, while real-world graphs are usually dynamically changing. However, existing index-update schemes can not achieve a sub-linear update time.

Motivated by this, we present an efficient indexing scheme for single-source PPR queries on evolving graphs. Our proposed solution is based on a classic framework that combines the forward-push technique with a random walk index for approximate PPR queries. Thus, our indexing scheme is similar to existing solutions in the sense that we store pre-sampled random walks for efficient query processing. One of our main contributions is an incremental updating scheme to maintain indexed random walks in expected $O(1)$ time after each graph update. To achieve $O(1)$ update cost, we need to maintain auxiliary data structures for both vertices and edges. To reduce the space consumption, we further revisit the sampling methods and propose a new sampling scheme to remove the auxiliary data structure for vertices while still supporting $O(1)$ index update cost on evolving graphs. Extensive experiments show that our update scheme achieves orders of magnitude speed-up on update performance over existing index-based dynamic schemes without sacrificing the query efficiency.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**.

Additional Key Words and Phrases: Personalized PageRank, evolving or dynamic graphs

ACM Reference Format:

Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme. *Proc. ACM Manag. Data* 1, 1, Article 25 (May 2023), 26 pages. https://doi.org/10.1145/3588705

*Sibo Wang is the corresponding author.

Authors' addresses: Guanhao Hou, ghhou@se.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Qintian Guo, qtguo@se.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Fangyuan Zhang, fzhang@se.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Sibo Wang, swang@se.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Zhewei Wei, zhewei@ruc.edu.cn, Renmin University of China, Beijing, China.

2836-6573/2023/5-ART25 $15.00
https://doi.org/10.1145/3588705

1 INTRODUCTION

Given an input graph G, a decay probability α, a source node s, and a target node t, the *personalized PageRank (PPR)* score $\pi(s,t)$ stands as a classic proximity measure of the relative importance of t from the viewpoint of s. More formally, the PPR score $\pi(s,t)$ of t with respect to s is the probability that an α-decay random walk [21] starting from s stops at t. Here, an *α-decay random walk* works as follows: it starts from the source s and at each step, it either *(i)* stops at the current node v (initially $v = s$) with probability α; or *(ii)* randomly jumps to one of the out-neighbors of v with probability $1-\alpha$ (specially, a self-loop will be applied to v if it has no out-neighbor). An important type of PPR queries is the *single-source PPR (SSPPR)* query where a source s is given, and the goal is to compute the PPR score $\pi(s,v)$ for each node v in the input graph G. SSPPR has many important applications in web search [21, 32], spam detection [19], community detection [6], social recommendations [18, 24], etc. Moreover, as a fundamental proximity measure, SSPPR is also widely used in graph representation learning, e.g., HOPE [31], Verse [36], STRAP [46], NRP [45], PPRGo [8], DynamicPPE [17], Lemane [48], and Tree-SVD [10].

With its wide applications, it is important to have efficient algorithms for SSPPR queries. However, computing exact answers for SSPPR queries is prohibitive on massive graphs. Existing state-of-the-art solutions [23, 44] all focus on *approximate SSPPR (ASSPPR)* queries with guarantees (Ref. to Definition 2.1). With the framework which combines the Forward-Push technique with the Monte-Carlo method for random walks, existing solutions, e.g., FORA+ [41] and SpeedPPR+ [44], pre-store random walks and show superb query efficiency on ASSPPR queries while providing strong approximation guarantees. However, all these index-based methods assume that the input graph is static while most real-world graphs are dynamically evolving. Social recommendation systems, e.g. Twitter Whom-to-Follow [18] and Tencent social recommendation [24] are virtually required to work on social networks which keep evolving frequently. Besides, some graph representation learning frameworks, e.g. SDG [12], need to estimate SSPPR on evolving graphs. Despite that index-free solutions can also handle ASSPPR queries on dynamic graphs, existing solutions with random walk index have a significant improvement in query performance compared to index-free solutions. In addition, early index-based solutions for ASSPPR queries on dynamic graphs [7, 47] have a prohibitive cost to maintain index structures. Even the state-of-the-art index-update solution, Agenda [29], still has a time consumption that is linearly related to the graph size for each update.

Motivated by limitations of existing solutions, we propose an efficient index scheme FIRM[1], which solves single-source PPR queries on evolving graphs. Our proposed solution is also based on the classic Forward-Push + Monte-Carlo framework, and thus it can provide query processing as efficiently as existing solutions on static graphs. One of our main contributions is that we design an incremental index update scheme to maintain the random walk index after each graph update in expected $O(1)$ time. The main idea is to find affected random walks for the recent graph update and adjust such random walks to fit the updated graph. We design auxiliary structures to trace affected random walks efficiently and achieve the $O(1)$ expected update time. In particular, we maintain auxiliary data structures for each node and each edge which record the information of random walks that go across each node and each edge, respectively. To reduce the space consumption, we further revisit the sampling method and present a non-trivial new sampling scheme to remove the requirement of the auxiliary data structure for each node. Remarkably, the new scheme comes at no sacrifice of the update time complexity as it still provides $O(1)$ expected time cost for each graph update. Extensive experiments show that our update scheme achieves up to orders of magnitude

[1] Forward-Push with Incremental Random Walk Maintenance

speed-up over existing index-based schemes on evolving graphs without sacrificing the query efficiency over index-based solutions.

2 PROBLEM DEFINITION

Let $G = (V, E)$ be a directed graph with node number n=$|V|$ and edge number m=$|E|$. For an edge e=$\langle u, v\rangle \in E$, we say that e is an outgoing edge of node u, and node v is an out-neighbor of node u. Then for a node $u \in V$, the set $\mathcal{N}(u)$ of its out-neighbors, the set $\mathcal{E}(u)$ of its outgoing edges, and the out-degree $d(u)$ of node u are denoted as $\mathcal{N}(u) = \{v | \langle u, v\rangle \in E\}$, $\mathcal{E}(u) = \{\langle u, v\rangle \in E\}$, and $d(u)$=$|\mathcal{E}(u)|$, respectively.

Personalized PageRank (PPR) [21]. Given a directed graph $G = (V, E)$, a source node $s \in V$ and a decay probability α, the PPR score of node t with respect to node s is defined as the probability that an α-decay random walk starting from s stops at t, denoted as $\pi(s, t)$. We use a vector $\boldsymbol{\pi}(s)$ to represent the PPR scores of all nodes in the graph with respect to source node s.

Approximate Single-Source PPR (ASSPPR). Following [29, 42, 44], this paper focuses on the ASSPPR problem defined as follows:

Definition 2.1 ((ϵ, δ)-ASSPPR). Given a threshold δ>0, a relative error 1>ϵ>0, and a failure probability p_f, an ASSPPR query returns an estimated PPR score $\tilde{\pi}(s, v)$ for each node $v \in V$ so that:

$$|\pi(s, v) - \tilde{\pi}(s, v)| < \epsilon \cdot \pi(s, v) \quad \forall \pi(s, v) \geq \delta$$

with at least $1-p_f$ probability where the parameter ϵ is usually in $(0, 1)$ and the parameters δ and p_f are typically set to $1/n$. □

Besides, many applications, e.g. recommendation system [18, 24] and graph neural networks [8, 12], are not interested in the full PPR vector with respect to s. Instead, they only use the vertices which have the highest PPR scores with respect to s. Thus, we study another type of PPR queries, called *approximate single-source top-k PPR (ASSPPR top-k)* queries [42] defined as follows:

Definition 2.2. ((ϵ, δ)-ASSPPR top-k) Given a threshold δ>0, a relative error 1>ϵ>0, a failure probability p_f, and a positive integer k, an ASSPPR top-k query returns a sequence of k nodes, $v_1, v_2, \ldots, v_k$, such that for any $i \in [1..k]$, the following equations hold with at least $1-p_f$ probability:

$$\begin{aligned} &|\pi(s, v_i) - \tilde{\pi}(s, v_i)| \leq \epsilon \cdot \pi(s, v_i) \\ &\pi(s, v_i) \geq (1 - \epsilon) \cdot \pi(s, v_i^*) \end{aligned} \quad \forall \pi(s, v_i^*) \geq \delta,$$

where v_i^* has the i-th largest exact PPR score with respect to s. □

Evolving Graph. We make a consistent assumption of the dynamically evolving graph with previous works [7, 29, 47]. There exist an initial graph at the beginning, followed by a sequence of updates. In this paper, we only consider edge updates: *edge insertion* and *edge deletion*. There may also exist node insertions/deletions on evolving graphs, both of which can be easily converted to a sequence of edge insertions/deletions. We add a subscript to specify which timestamp we are discussing. Let G_0 be the initial graph, e_τ denotes the τ-th updating edge (the update is either an insertion or a deletion). Let $G_\tau = (V_\tau, E_\tau)$ denotes the graph after the τ-th update has been applied. Note that in the insertion case, we have $E_\tau = E_{\tau\text{-}1} \cup \{e_\tau\}$; in the deletion case, we have $E_\tau = E_{\tau\text{-}1} \setminus \{e_\tau\}$. Besides, we consider that the τ-th update occurs at timestamp τ, and thus the update at timestamp τ refers to the τ-th update.

With the edge updating model, we further assume that the sequence of updates is uniformly random, which is called *random arrival model* in [7, 47]. More precisely, it is defined as follows.

Notation	Description
$G = (V, E)$	a directed graph consists of node set V and edge set E
n, m	the number of nodes and edges, respectively
$\mathcal{N}(u)$	the set of out-neighbors of node u
$\mathcal{E}(u)$	the set of outgoing edges of node u
$d(u)$	the out-degree of node u
G_τ=(V_τ, E_τ)	the graph after the τ-th update has been executed
e_τ=$\langle u_\tau, v_\tau \rangle$	the τ-th updating edge, which occurs at timestamp τ
$\pi(s, v)$	the PPR score for node v with respect to source node s
$H, H(s)$	the pre-stored random walk index and those starting from node s, respectively
$\tilde{\pi}(s, v)$	the estimation of the PPR score $\pi(s, v)$
$\hat{\pi}(s, v), r(s, v)$	the reserve and residue value of node v with respect to source node s, respectively
ϵ, δ, p_f	relative error bound, effective threshold, and failure probability, respectively

Table 1. Frequently used notations.

Definition 2.3 (Random arrival model). For edge insertion, the probability that e= $\langle u, v \rangle$ is inserted at timestamp τ is

$$\mathbb{P}\left[e_\tau = e\right] = \frac{1}{m_\tau},$$

i.e., the probability that each edge in E_τ is the last one to be inserted at timestamp τ is equal. In the case of edge deletion, the probability that any edge e= $\langle u, v \rangle \in E_{\tau\text{-}1}$ will be deleted at timestamp τ is

$$\mathbb{P}\left[e_\tau = e\right] = \frac{1}{m_{\tau\text{-}1}},$$

i.e., each edge has equal probability to be deleted at timestamp τ. □

From the above definition, we could immediately know that the probability that the updating edge e_τ is an outgoing edge of node u in the edge insertion case is

$$\mathbb{P}\left[e_\tau \in \mathcal{E}_\tau(u)\right] = \frac{d_\tau(u)}{m_\tau}, \tag{1}$$

and for the edge deletion case, the corresponding probability is

$$\mathbb{P}\left[e_\tau \in \mathcal{E}_{\tau\text{-}1}(u)\right] = \frac{d_{\tau\text{-}1}(u)}{m_{\tau\text{-}1}}. \tag{2}$$

In the rest of this paper, for the sake of brevity, we use the non-subscripted notations (e.g., G, E, d) in the context if there is no danger of confusion. Otherwise, the subscripted notations are used. In Table 1, we list the notations frequently used in this paper.

3 EXISTING SOLUTIONS

In this section, we first revisit two index-free solutions for ASSPPR queries on static graphs: FORA [41] and SpeedPPR [44] and their index-based solution FORA+ and SpeedPPR+. Then, we review Agenda [29], the state-of-the-art solution for evolving graphs.

3.1 Solutions on Static Graph

FORA. Wang et al. [41] propose a two-phase solution, dubbed as FORA, to answer ASSPPR queries. It first performs the Forward-Push technique [6] as Algorithm 1 shows. In particular, it maintains two vectors, the reserve vector $\hat{\boldsymbol{\pi}}(\boldsymbol{s})$ and the residue vector $\boldsymbol{r}(\boldsymbol{s})$. Initially, the vector $\hat{\boldsymbol{\pi}}(\boldsymbol{s})$ is set to zero on all entries and we use $\hat{\pi}(s, v)$ to denote the v-th entry of vector $\hat{\boldsymbol{\pi}}(\boldsymbol{s})$. For the residue vector $\boldsymbol{r}(\boldsymbol{s})$, it is initialized as $\mathbf{1}_s$ where $\mathbf{1}_s$ denotes the one-hot vector with respect to s, i.e., only $r(s, s)$

Algorithm 1: Forward-Push

Input: Graph $G = (V, E)$, decaying rate α, threshold r_{max}, source node s;
Output: reserve vector $\hat{\boldsymbol{\pi}}(s)$ and residue vector $\boldsymbol{r}(s)$;

1 $\hat{\boldsymbol{\pi}}(s) \leftarrow \mathbf{0}, \boldsymbol{r}(s) \leftarrow \mathbf{1}_s$;
2 **while** $\exists u \in V$ **such that** $\frac{r(s,u)}{d(u)} \geq r_{max}$ **do**
3 $\quad \hat{\pi}(s,u) \leftarrow \hat{\pi}(s,u) + \alpha \cdot r(s,u)$;
4 $\quad$ **foreach** $v \in \mathcal{N}(u)$ **do**
5 $\quad\quad r(s,v) \leftarrow r(s,v) + (1-\alpha) \cdot \frac{r(s,u)}{d(u)}$;
6 $\quad$ **end**
7 $\quad r(s,u) \leftarrow 0$;
8 **end**
9 **return** $[\hat{\boldsymbol{\pi}}(s), \boldsymbol{r}(s)]$;

is 1 and other positions are zero. Then for any node u with residue no smaller than $r_{max} \cdot d(u)$, it performs a *push operation* to u, which converts α portion of the current residue to the reserve of u (Line 3), and for remaining residues, they are evenly propagated to out-neighbors of u (Lines 4-5). It stops when no node satisfies the push condition (Line 2). After the Forward-Push phase, it simulates sufficient random walks to give the final result. The rationale of FORA comes from the following invariant which always holds during the Forward-Push phase:

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

FORA computes $\hat{\pi}(s,t)$ to roughly approximate the PPR score in the first phase and then exploits random walks to estimate the cumulative term $\sum_{v \in V} r(s,v) \cdot \pi(v,t)$ in the second phase, thus refining the estimation. The following lemma about the number of random walks is proved in [41].

Lemma 3.1. *Given the threshold of residue r_{max}, the number of independent random walks starting from node $v \in V$ should be at least $\lceil r(s,v) \cdot \omega \rceil$ to satisfy (ϵ, δ)-approximate guarantee, where*

$$\omega = \frac{((2/3) \cdot \epsilon + 2) \cdot \log(2/p_f)}{\epsilon^2 \delta}. \tag{4}$$

As [6] shows that the time complexity of Forward-Push phase corresponding to the threshold of residue r_{max} is $O(1/r_{max})$, FORA sets r_{max}=$\sqrt{1/(m \cdot \omega)}$ to optimize the time complexity to $O(\sqrt{m \cdot \omega})$. Since a scale-free graph with $\gamma \in [2,3]$ has the average degree $\bar{d}$=m/n=$O(\log n)$, the above time complexity will be $O(n \cdot \log n/\epsilon)$ with δ=$1/n$ and p_f=$1/n$. Interested readers can refer to [41] for more details of the time complexity.

FORA+. Wang et al. [41] also propose an index scheme of FORA that works on static graphs, dubbed as FORA+, which pre-computes a sufficient number of random walks to further improve the query efficiency of their algorithm. For each random walk, FORA+ simply stores the source and terminal node. Then, when FORA+ needs to sample a random walk starting from a node u, it directly gets a pre-stored random walk with u as the source and chooses a pre-stored terminal node. Notice that each pre-stored random walk will be used at most once for an ASSPPR query to guarantee that random walks are independent from each other. One important question for FORA+ is how many random walks should be pre-computed for each node $v \in V$ such that it could achieve both accuracy guarantee and space efficiency. According to Lemma 3.1, they have:

Lemma 3.2. *Given the threshold of residue r_{max}, the number of pre-computed independent random walks starting at node $v \in V$ should be at least $\lceil d(v) \cdot r_{max} \cdot \omega \rceil$ to satisfy (ϵ, δ)-approximate guarantee.*

By setting r_{max}=$\sqrt{1/(m\cdot\omega)}$ as in FORA, FORA+ [41] has a space complexity of $\Theta(m+\sqrt{m\cdot\omega})$ and will become $O(n\cdot\log n/\epsilon)$ on scale-free graphs with $\gamma\in[2,3]$.

SpeedPPR. Wu et al. [44] propose an improved FORA version, called SpeedPPR, to answer ASSPPR queries. In practice, SpeedPPR takes benefits from their cache-friendly implementation of Forward-Push, called Power-Push, which combines the power iteration and vanilla Forward-Push into a whole. As well as they theoretically prove that the Forward-Push phase could achieve a time complexity of $O(m\cdot\log(1/r_{max}))$ apart from the previously known result $O(1/r_{max})$ and this bound improves when r_{max} is very small, i.e., $O(1/m)$. With the aid of such new knowledge, they further set the threshold r_{max}=$\Theta(1/\omega)$ in SpeedPPR, rather than $\sqrt{1/(m\cdot\omega)}$ in the original FORA. It is proved in [44] that such a modification yields a time complexity of $O(n\cdot\log n\cdot\log(1/\epsilon))$ on scale-free graphs, improving a factor of $1/(\epsilon\cdot\log(1/\epsilon))$ over FORA. Interested readers can refer to [44] for more details of the time complexity.

SpeedPPR+. SpeedPPR also admits an index-based version called SpeedPPR+ [44]. It is shown in [44] that as $r_{max}\cdot\omega$=$\Theta(1)$, the index size of SpeedPPR+ is $\Theta(m)$, independent of the relative error ϵ and effective threshold δ, and smaller than the space complexity of FORA+, which is $\Theta(m+\sqrt{m\cdot\omega})$.

3.2 Solutions on Evolving Graph

There is no doubt that pure-online approaches such as FORA can be applied to dynamically evolving graphs directly. Nevertheless, the previous works [41, 44] show that an index-based scheme is much more efficient in query processing than its index-free version. On the other hand, there is also a trivial tactic to adapt indexing schemes to evolving graphs which reconstructs the whole index after every update. However, such a strategy is obviously inefficient in dynamic graphs while most real-world graphs evolve frequently.

The early index-based solutions for ASSPPR queries on evolving graphs try to maintain random walks [7] or the reserve and residue vectors [47] incrementally, while the index cost and update cost are still inconceivable to answer ASSPPR queries for arbitrary source nodes. For instance, the solution in [7] needs $O(n^2)$ space to pre-store random walks to answer ASSPPR queries for arbitrary source nodes when δ=$1/n$, making them infeasible to large graphs.

Agenda. Mo and Luo [29] propose a feasible approach to maintain the index of FORA on evolving graphs named Agenda, which aims to balance the query and update efficiencies of ASSPPR estimation. The core strategy of Agenda is lazy-update, which has a tolerance for the inaccuracy of random walks and reconstructs a part of random walks when the error exceeds the limit.

Agenda introduces a parameter $\theta\in(0,1)$ and makes an inaccuracy tolerance of its index. When an edge update e_τ=$\langle u_\tau, v_\tau\rangle$ comes, it traces the (upper bound of) inaccuracy of current index by performing a Backward-Push [5] starting from u_τ, and accumulates the inaccuracy of each node into a vector $\boldsymbol{\sigma}$. For query processing, Agenda splits the error tolerance. Specifically, it first invokes the Forward-Push phase of FORA whereas the r_{max} is set according to the relative error bound $\theta\cdot\epsilon$ instead of ϵ. With this tightened error bound $\theta\cdot\epsilon$, if there is no update at all, the query accuracy of Agenda tends to be higher than that of the original FORA/SpeedPPR. Then, it checks the query-dependent inaccuracy of current index with $\boldsymbol{e}$=$\boldsymbol{\sigma}\circ\boldsymbol{r}$ (where $\boldsymbol{r}$ is the residue vector and $\circ$ is the pairwise multiplication). Next, it repeats to reconstruct all random walks starting from node v which has the largest value in $\boldsymbol{e}$, until the inaccuracy of the index with respect to the query will not exceed a relative error $(1-\theta)\cdot\epsilon$. Finally, Agenda enters the refining phase of FORA with the less inaccurate index. Since the relative error of the two parts (i.e. the FORA process and the index itself) is bounded by $\theta\cdot\epsilon$ and $(1-\theta)\cdot\epsilon$, respectively, the final result can be bounded by relative error ϵ, so that Agenda can answer an ASSPPR query with (ϵ,δ)-approximate guarantee.

Algorithm 2: Update-Insert

Input: Graph $G_\tau = (V_\tau, E_\tau)$, past index $H_{\tau\text{-}1}$, edge $e_\tau = \langle u_\tau, v_\tau \rangle$;
Output: New index H_τ and C^V_τ;

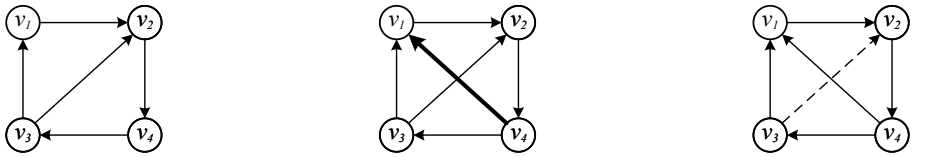
$H_\tau \leftarrow H_{\tau\text{-}1}$;
$C \leftarrow$ **Sample**$(C^V_{\tau\text{-}1}(u_\tau), \frac{1}{d_\tau(u_\tau)})$;
foreach $c \in C$ **do**
  $w \leftarrow H_\tau[c.id]$;
  $w[c.step{+}1] \leftarrow v_\tau$;
  $w \leftarrow$ **Walk-Restart**$(G_\tau, w, c.step{+}1)$;
end
while $|H_\tau(u_\tau)| < \lceil d_\tau(u_\tau) \cdot r_{max} \cdot \omega \rceil$ **do**
  $H_\tau \leftarrow H_\tau \cup \{$**Random-Walk**$(G_\tau, u_\tau)\}$;
end
return H_τ;

It is proved in [29] that the expected fraction of index to be reconstructed for each update will be either $O(\epsilon \cdot \bar{d}_\tau / \log n_\tau)$ by setting r^b_{max}=$\Theta(1/n_\tau)$ on directed graphs (where $\bar{d}_\tau$ is the average degree of G_τ), or $O(\epsilon / \log n_\tau)$ by setting r^b_{max}=$\Theta(d_\tau(u_\tau)/m_\tau)$ on undirected graphs. Agenda significantly reduces the number of random walks to be re-sampled for each update, whereas according to [5], backward-push expectantly runs within $O(\bar{d}/r^b_{max})$ time for random starting node, and thus the time cost for tracing the inaccuracy is $O(m_\tau)$ for each update, which will also become more and more expensive when the graph is becoming larger and larger.

Agenda$^\#$. Since Agenda divides the error tolerance into two parts, the error bound of its FORA process becomes $\theta \cdot \epsilon$ which is tighter than that of the original FORA/SpeedPPR. That means Agenda requires a more precise result of Forward-Push phase, and thus it makes a trade-off between query performance and update cost. On the other hand, Agenda makes a conservative estimation to bound the error resulting from the remaining non-updated part of the index, which is $(1-\theta)\cdot\epsilon$. In practice, the actual inaccuracy of its index after the lazy-update process often becomes far smaller than the upper bound it reckons. Therefore, in most cases, Agenda wastes computation resources to provide over-exquisite results. In our experiments, we present a new version of Agenda, dubbed as Agenda$^\#$, which aggressively assumes that the inaccuracy (with respect to the processing query) after the lazy-update phase is negligible. Thus, We set the relative error of the FORA process to be ϵ instead of $\theta \cdot \epsilon$. By such a strategy, the worst case relative error of Agenda$^\#$ becomes $(2-\theta)\cdot\epsilon$. As we will see, Agenda$^\#$ provides an accuracy pretty close to that of FORA/FORA+ when FORA/FORA+ provides a worst-case ϵ-relative error. We include Agenda$^\#$ as our baseline since we aim to provide a fair comparison where we compare the efficiency when all methods provide a similar accuracy.

4 THE INCREMENTAL APPROACH

Real-world graphs are usually dynamically evolving, which motivates us to find an efficient algorithm for ASSPPR problems on evolving graphs. Unfortunately, we need to make a great effort to achieve this goal. On one hand, the index-free solutions for static graphs, i.e., SpeedPPR, could easily handle the dynamic scenarios on their own right, but they may suffer from query efficiency issues since they need to perform a number of random walks for each query. On the other hand, as aforementioned, existing index-based approaches for ASSPPR problems on evolving graphs, have a notable time cost to maintain their index. Even the state-of-the-art solution, Agenda, needs a time linearly correlating to the graph size for each update. Since many real-world graphs, especially



(a) Initial graph G_0 (b) G_1 after insertion (c) G_2 after deletion

Fig. 1. A Tiny Evolving Graph.

the social networks are colossal, there will be a prohibitively computational overhead for index maintenance if update events happen frequently (unluckily, it is the usual case). Therefore, it deserves our effort to design a better index-based solution to tackle the challenges.

Solution overview. Generally speaking, our goal is also to find an efficient way to maintain the index of FORA+. However, different from Agenda, our proposal uses an eager-update strategy. Compared to the lazy-update strategy used by Agenda, our solution has a query efficiency almost the same as the solutions on static graphs because we make no trade-off between update and query efficiency. More importantly, by carefully tracing and minimally adjusting the random walks affected by each update, our solution has a constant time cost per update in expectation so that it can be applied to massive-scale and highly-frequently evolving graphs.

To achieve efficient tracing, we should maintain more information on each random walk instead of its starting and terminal nodes. More specifically, each element w of our index (denoted as H) is a completed path $\langle v_0, v_1, ..., v_l \rangle$ starting from v_0, crossing $v_1, ..., v_{l-1}$ in order, and terminating at v_l. Besides, we use $H(u)$ to denote the subset of H that contains all random walks starting from u. Next, we will introduce details of our solutions, which take an expected constant index update time for both edge insertion and deletion.

Remark. As a node without any incident edge on it has no effect on PPR values of other nodes, we can handle the node insertion case in this way: a new node is automatically inserted, exactly when the first edge which is incident on the new node arrives. Then, we proceed with remaining added edges following our edge insertion algorithm. In the case of node deletion, we can delete a node by deleting all edges incident to the node one by one, and the node will be deleted automatically once the last incident edge is removed.

4.1 Edge Insertion

Intuitively, assuming that an edge $\langle u_\tau, v_\tau \rangle$ is inserted, we need to update the pre-stored random walks to guarantee that they are sampled according to the current graph structure. Meanwhile, the insertion only makes a slight change to the graph structure and hence it should affect only slightly the random walks. More specifically, we will show that if a pre-stored random walk does not cross node u_τ, then the random walk does not need to be updated. Notice that here we only consider random walks that cross u_τ and do not include random walks that only terminate at u_τ (but have not crossed u_τ). A quick explanation is that if the random walk just stops at node u_τ, which means it does not cross u_τ, it is still not affected as it is not dependent on the newly inserted edge $\langle u_\tau, v_\tau \rangle$.

Algorithm 2 shows the pseudo-code of our **Update-Insert** algorithm to handle index update with an edge insertion. The symbol C^V denotes an auxiliary data structure of index H. Each element $c=\{id, step\} \in C^V(u)$ is a crossing record, representing that the random walk with ID $c.id$ crosses u at step $c.step$. We update $H_{\tau\text{-}1}$ to H_τ after the insertion of $e_\tau = \langle u_\tau, v_\tau \rangle$ by *(i)* sampling records in $C^V(u_\tau)$ with probability $1/d_\tau(u_\tau)$; *(ii)* updating sampled random walks on the evolved graph. In particular, for each sampled record c from $C^V_{\tau\text{-}1}(u_\tau)$, we first redirect the random walk with ID $c.id$ to the newly inserted edge e_τ at its ($c.step$+1)-th step (Line 5), and then make it continue to

	ID: walking path
$H_0(v_1)$	$1 : v_1 \rightarrow v_2 \rightarrow v_4,\ 2 : v_1 \rightarrow v_2$
$H_0(v_2)$	$3 : v_2 \rightarrow v_4,\ 4 : v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow v_1$
$H_0(v_3)$	$5 : v_3 \rightarrow v_1 \rightarrow v_2,\ 6 : v_3 \rightarrow v_2 \rightarrow v_4,\ 7 : v_3 \rightarrow v_1$
$H_0(v_4)$	$8 : v_4 \rightarrow v_3 \rightarrow v_1,\ 9 : v_4 \rightarrow v_3$

(a) Index structure H_0

	record $c = \{id, step\}$
$C_0^V(v_1)$	$\{1,0\},\ \{2,0\},\ \{5,1\}$
$C_0^V(v_2)$	$\{1,1\},\ \{3,0\},\ \{4,0\},\ \{6,1\}$
$C_0^V(v_3)$	$\{4,2\},\ \{5,0\},\ \{6,0\},\ \{7,0\},\ \{8,1\}$
$C_0^V(v_4)$	$\{4,1\},\ \{8,0\},\ \{9,0\}$

(b) Auxiliary structure C_0^V

Table 2. An example of the index and auxiliary structures.

randomly traverse until it has the same hops as the random walk with ID $c.id$ in $H_{\tau\text{-}1}$ (Line 6). After updating existing random walks, we further add additional random walks starting at u_τ into H_τ (Lines 8-10) to increase the size of $H_\tau(u_\tau)$ to fulfill conditions in Lemma 3.2 and finish the update.

Example 4.1. Consider an original graph G_0 as shown in Fig. 1(a), then assume that the initial index structure H_0 and auxiliary structures C_0^V are as shown in Table 2. In Table 2(a), the row labeled with $H_0(v_i)$ stores the random walks which start from v_i. For example, there are two random walks (which have IDs 1 and 2) starting from v_1. To simplify the description, we use $\mathrm{RW_i}$ as the representation of the random walk with ID i. In Table 2(b), the row labeled with $C_0^V(v_i)$ records which random walks visit node v_i. In the case as Table 2 shows, there are three records: $\{1,0\}, \{2,0\}, \{5,1\}$ in the row corresponding to v_1. The record of $\{5,1\}$ represents that $\mathrm{RW_5}$ visits v_1 at its 1st step, which can be verified by the detail of $\mathrm{RW_5}$ showed in Table 2(a). Now, when an edge insertion $\langle v_4, v_1 \rangle$ comes as shown in Fig. 1(b), the index will be updated by following steps:

(i) Sample random walks in $C_0^V(v_4)$ with probability 1/2 (as the out-degree of v_4 at timestamp 1 is 2) and then add sampled random walks to C. We assume that $\mathrm{RW_4}$ and $\mathrm{RW_8}$ are chosen to be updated as $C_0^V(v_4)$ contains records $\{4,1\}, \{8,0\}, \{9,0\}$.

(ii) Redirect $\mathrm{RW_4}$ to v_1 at its 2nd step and re-sample from its 3rd step, since the only out-neighbour of v_1 is v_2, $\mathrm{RW_4}$ will visit v_2 at step 3. Then $\mathrm{RW_4}$ has a length of 3 that is equal to the number of hops it held in H_0, so it will stop. A similar process is applied to $\mathrm{RW_8}$ as well. Table 3(a) shows a possible repaired index where the adjusted part is shown in boldface.

(iii) Add one more random walk (underlined) to fulfill the requirement as the out-degree of v_4 is incremented by 1.

(iv) C^V gets updated to reflect changes of $\mathrm{RW_4}$, $\mathrm{RW_8}$ and the new random walk $\mathrm{RW_{10}}$. For example, $\{8,1\}$ in $C_0^V(v_3)$ is moved into $C_1^V(v_1)$. Table 3(b) shows the updated C_1^V, where new records are in boldface and obsoleted records are scratched. □

To show the correctness of **Update-Insert**, we first make a review of FORA-like approaches in Section 3.1. Such methods roughly approximate PPR scores in Forward-Push phase and then refine the estimation using the sampled random walks. Thus, we have:

Lemma 4.2. *An index-based FORA-like solution gives a guarantee of (ϵ, δ)-approximation for the result if the following conditions hold:*

- **Independence.** *The random walks in H are independent.*

$H_1(v_1)$	$1 : v_1 \rightarrow v_2 \rightarrow v_4,\ 2 : v_1 \rightarrow v_2$
$H_1(v_2)$	$3 : v_2 \rightarrow v_4,\ 4 : v_2 \rightarrow v_4 \rightarrow \boldsymbol{v_1} \rightarrow \boldsymbol{v_2}$
$H_1(v_3)$	$5 : v_3 \rightarrow v_1 \rightarrow v_2,\ 6 : v_3 \rightarrow v_2 \rightarrow v_4,\ 7 : v_3 \rightarrow v_1$
$H_1(v_4)$	$8 : v_4 \rightarrow \boldsymbol{v_1} \rightarrow \boldsymbol{v_2},\ 9 : v_4 \rightarrow v_3,\ \underline{10 : v_4 \rightarrow v_3}$

(a) The index structure H_1

$C_1^V(v_1)$	$\{1,0\},\ \{2,0\},\ \mathbf{\{4,2\}},\ \{5,1\},\ \mathbf{\{8,1\}}$
$C_1^V(v_2)$	$\{1,1\},\ \{3,0\},\ \{4,0\},\ \{6,1\}$
$C_1^V(v_3)$	~~$\{4,2\}$~~, $\{5,0\},\ \{6,0\},\ \{7,0\},$ ~~$\{8,1\}$~~
$C_1^V(v_4)$	$\{4,1\},\ \{8,0\},\ \{9,0\},\ \mathbf{\{10,0\}}$

(b) The updated entries in auxiliary structure C_1^V

Table 3. Updated index and C^V after Insertion.

- **Adequateness.** *For each node s, the set $H(s)$ of pre-stored random walks starting from s satisfies Lemma 3.2.*
- **Unbiasedness.** *Let $H(s,t)$ denote the subset of $H(s)$ where $w \in H(s,t)$ means w starting from s terminates at t. For arbitrary nodes $s, t \in V$, $|H(s,t)|/|H(s)|$ is an unbiased estimation of $\pi(s,t)$.*

Proof Sketch. The adequateness condition directly comes from Lemma 3.2, and Lemma 3.2 is derived from a concentration bound (more precisely, Theorem 5 in [9]), of which the independence and unbiasedness conditions are preconditions. We need to meet these conditions so that the accuracy of PPR scores can be guaranteed. □

Hence, if we build an initial index H_0 which is directly sampled independently on G_0 that contains a sufficient number of random walks from each node to satisfy the above conditions, the correctness of **Update-Insert** can be derived from the following result:

Theorem 4.3. *H_τ provided by Algorithm 2 satisfies the independence, adequateness, and unbiasedness if $H_{\tau\text{-}1}$ holds these properties.*

The above results indicate that the updated index still satisfies three conditions and provides the approximation guarantee. Notably, the update cost to the random walk index is very light and can be bounded with $O(1)$ in expectation as shown in Theorem 4.4.

Theorem 4.4. *Given an inserted edge that follows the random arrival model (Definition 2.3), Algorithm 2 takes $O(1)$ expected time to update the index when $r_{max} \cdot \omega = \Theta(1)$.*

For ease of exposition, all proofs are deferred to Section 5.

4.2 Edge Deletion

When an edge deletion to $\langle u_\tau, v_\tau \rangle$ occurs, the main idea to update the random walks with an edge deletion is similar to that of edge insertion, by updating the affected random walks that cross u_τ.

Algorithm 3 shows the pseudo-code of **Update-Delete**. We use the symbol C^E to denote the auxiliary structure of H where each $c \in C^E(e)$ is a crossing record where $c=\{id, step\}$ denotes the random walk $c.id$ passes through e between step $c.step$ and $c.step$+1. Note that we require a smaller number of random walks starting from u_τ after the deletion of $e_\tau = \langle u_\tau, v_\tau \rangle$. That means some random walks in $H_{\tau\text{-}1}(u_\tau)$ are no longer needed. To avoid the waste of space to store the index, we uniformly select some random walks $w^* \in H_{\tau\text{-}1}(u)$ and remove them before adjusting the index (Lines 3-7). Then, for each record $c \in C^E_{\tau\text{-}1}(e_\tau)$ (except records corresponding to recently trimmed random walks, denoted as $C^E_{\tau\text{-}1}(W^*)$), the corresponding random walk $c.id$ must be repaired since

Algorithm 3: Update-Delete

Input: Graph $G_\tau = (V_\tau, E_\tau)$, past index $H_{\tau\text{-}1}$, edge $e_\tau = \langle u_\tau, v_\tau \rangle$;
Output: New index H_τ;

1 $H_\tau \leftarrow H_{\tau\text{-}1}$;
2 $W^* \leftarrow \emptyset$;
3 **while** $|H_\tau(u_\tau)| > \lceil d_\tau(u_\tau) \cdot r_{max} \cdot \omega \rceil$ **do**
4 $\quad w^* \leftarrow$ **Sample-Uniform**$(H_\tau(u_\tau))$;
5 $\quad H_\tau \leftarrow H_\tau \setminus \{w^*\}$;
6 $\quad W^* \leftarrow W^* \cup \{w^*\}$
7 **end**
8 **foreach** $c \in C^E_{\tau\text{-}1}(e_\tau) \setminus C^E_{\tau\text{-}1}(W^*)$ **do**
9 $\quad w \leftarrow H_\tau[c.id]$;
10 $\quad H_\tau[c.id] \leftarrow$ **Walk-Restart**$(G', w, c.step)\}$;
11 **end**
12 **return** H_τ;

e_τ no longer exists in G_τ. Let such a random walk restart at step $c.step$ where it hits node u_τ before passing through edge e_τ (Lines 8-11). Then, it randomly traverses on G_τ until it has the same hops as the $c.id$ random walk in $H_{\tau\text{-}1}$. For other random walks, we keep them as they are in $H_{\tau\text{-}1}$.

Example 4.5. Assume that the current graph is G_1 as shown in Figure 1(b) after the insertion in Example 4.1. Then, when the deletion of edge $\langle v_3, v_2 \rangle$ comes, it will evolve to G_2 as shown in Figure 1(c). To simplify the description, we omit the auxiliary structure C^V on nodes as it is not used in deletion and it will be maintained in the same way as we described in Example 4.1. After the edge deletion, we will update the index with the following steps:

(i) Remove one random walk in $H_2(v_3)$ to save space. We are allowed to do it as the out-degree of v_3 is decreased by 1. Assume that RW_7 is removed (crossed-out as shown in Table 4(a)).

(ii) Repair RW_6 since it passes through edge $\langle v_3, v_2 \rangle$ between its step 0 and step 1. We simulate the restarting part of RW_6 from its step 0. A possible repaired index is as shown in Figure 4(b) where the newly removed random walk is crossed-out and the repaired parts of existing random walks are marked in boldface. The other entries $H(\cdot)$ that are not changed are omitted.

(iii) To match the updated random walks, the auxiliary structure C^E needs to be updated accordingly. For example, since RW_6 is changed from $v_3 \rightarrow v_2 \rightarrow v_4$ to $v_3 \rightarrow v_1 \rightarrow v_2$, the records $\{6, 0\}$ and $\{6, 1\}$ are moved into $C^E_2(v_3, v_1)$ and $C^E_2(v_1, v_2)$ from $C^E_1(v_3, v_2)$ and $C^E_1(v_2, v_4)$, respectively. Table 4(c) shows the updated C^E_2 where the new records are boldfaced and the obsoleted records are scratched. Note that we also scratch the entry $C^E_2(v_3, v_2)$ in Table 4(c) as it is stale due to the deletion of edge $\langle v_3, v_2 \rangle$. □

Similar as we described in Section 4.1, the following result gives the correctness of **Update-Delete** since we can build an initial index H_0 satisfying independence, adequateness, and unbiasedness.

Theorem 4.6. *H_τ provided by Algorithm 3 satisfies the independence, adequateness, and unbiasedness if $H_{\tau\text{-}1}$ holds these properties.*

Just like the edge insertion case, the update cost will be light if an edge deletion follows the random arrival model. As Theorem 4.7 shows, the cost can be bounded by $O(1)$.

Theorem 4.7. *Given an edge deletion that follows the random arrival model (Definition 2.3), Algorithm 2 takes $O(1)$ expected time to update the index when $r_{max} \cdot \omega = \Theta(1)$.*

$C_1^E(v_1, v_2)$	$\{1,0\},\ \{2,0\},\ \{4,2\},\ \{5,1\},\ \{8,1\}$		
$C_1^E(v_2, v_4)$	$\{1,1\},\ \{3,0\},\ \{4,0\},\ \{6,1\}$		
$C_1^E(v_3, v_1)$	$\{5,0\},\ \{7,0\}$	$C_1^E(v_3, v_2)$	$\{6,0\}$
$C_1^E(v_4, v_1)$	$\{4,1\},\ \{8,0\}$	$C_1^E(v_4, v_3)$	$\{9,0\},\ \{10,0\}$

(a) Auxiliary Structure C_1^E

$H_2(v_3)$	$5 : v_3 \rightarrow v_1 \rightarrow v_2,\ 6 : v_3 \rightarrow \boldsymbol{v_1} \rightarrow \boldsymbol{v_2},\ \cancel{7 : v_3 \rightarrow v_1}$

(b) The updated index entries in H_2

$C_2^E(v_1, v_2)$	$\{1,0\},\ \{2,0\},\ \{4,2\},\ \{5,1\},\ \boldsymbol{\{6,1\}},\ \{8,1\}$		
$C_2^E(v_2, v_4)$	$\{1,1\},\ \{3,0\},\ \{4,0\},\ \cancel{\{6,1\}}$		
$C_2^E(v_3, v_1)$	$\{5,0\},\ \boldsymbol{\{6,0\}},\ \cancel{\{7,0\}}$	$\cancel{C_2^E(v_3, v_2)}$	$\cancel{\{6,0\}}$
$C_1^E(v_4, v_1)$	$\{4,1\},\ \{8,0\}$	$C_1^E(v_4, v_3)$	$\{9,0\},\ \{10,0\}$

(c) The updated entries in auxiliary structure C_2^E

Table 4. Index and Auxiliary Structures after Deletion.

For ease of exposition, all proofs are deferred to Section 5.

4.3 A New Sampling Scheme for Index Update

Up to now, we have shown the general framework of our solution which updates the index within constant time for each edge insertion/deletion. However, a simple implementation of Algorithm 2 and Algorithm 3 will result in several times space over existing solutions, if we maintain auxiliary structures in views of both nodes and edges. Next, we show a new sampling scheme for index updates to reduce the space consumption of auxiliary structures.

Main Challenge. As we described in Sections 4.1 and 4.2, our solution achieves superb update efficiency since the expected number of random walks affected by an edge update is $O(1)$. However, it means that we need to trace each affected random walk within $O(1)$ time as well. To achieve this, we maintain auxiliary structures in the view of nodes (denoted as C^V) in Algorithm 2 and in the view of edges (denoted as C^E) in Algorithm 3, and thus our solution requires several times space to save the index. Therefore, it deserves our effort to use only one auxiliary structure to support both sampling among random walks crossing a node u and finding all random walks passing through an updated edge $e = \langle u, v \rangle$.

Since we need to exactly find all random walks passing through the edge e_τ in Algorithm 3, it is difficult to use only C^V to achieve our goal. Thus, we turn to find an alternative sampling scheme for index update to sample random walks using C^E in Algorithm 2.

Before we introduce how to sample with C^E in the insertion case, there is an issue that a random walk starting from s maybe terminate directly with probability α so that it does not pass through any edge and therefore cannot be recorded by any edge. To fix the problem, consider a variant of Equation 3 derived in [41]:

$$\begin{aligned}\pi(s,t) &= \hat{\pi}(s,t) + \sum_{v \in V} r(s,v) \cdot \Big(\sum_{l=0}^{\infty} \pi^l(v,t)\Big) \\ &= \hat{\pi}(s,t) + \sum_{v \in V} r(s,v) \cdot (\pi^0(v,t) + \pi^+(v,t)),\end{aligned}$$

where $\pi^l(v,t)$ is the l-hop PPR which denotes the random walk starting from v stops at t exactly at its l-th step, and $\pi^+(v,t)$ is the sum of $\pi^l(v,t)$ for all non-zero l. Note that, $\pi^0(v,t)$=α only when v=t, and $\pi^0(v,t)$=0 otherwise. Because of the certainty of $\pi^0(v,\cdot)$, we need not store the random

Algorithm 4: Edge-Sampling

Input: Node u, record lists on edges C^E;
Output: Sampled records C;

```
C ← ∅;
k ~ B(c_τ(u_τ), 1/d_τ(u_τ));
for i from 1 to k do
    E ← Active-Edges(u_τ);
    e ← Sample-Uniform(E);
    c ← Sample-Uniform(C^E(e));
    C ← C ∪ {c};
end
return C;
```

walks which terminate at their source node immediately, and thus $C^E(e)$ is complete to tracing all random walks we need to maintain.

Recall that after the edge $e_\tau = \langle u_\tau, v_\tau \rangle$ is inserted, we need to sample each record crossing u_τ with probability $1/d_\tau(u_\tau)$. For the convenience of description, we will use the symbol C^V and C^E which have no subscript to represent $C^V_{\tau\text{-}1}$ and $C^E_{\tau\text{-}1}$ respectively, and then let $c^V(u)$ and $c^E(e)$ denote the size of $C^V(u)$ and $C^E(e)$ respectively. The naive way is to generate random numbers for every record in $C^V(u_\tau)$ to decide whether a record will be adjusted or not. This approach can be easily implemented with C^E where we can access the records in each $C^E(e)$ for $e \in \mathcal{E}_{\tau\text{-}1}(u_\tau)$ and then roll the dice. However, such a method uses $O(c^V(u_\tau))$ time which will be $O(d_{\tau\text{-}1}(u_\tau))$=$O(d_\tau(u_\tau))$ in expectation to select the random walks to be adjusted while only $O(1)$ random walks will be selected as we claimed in Section 4.1.

An alternative approach is geometric sampling. We number the records in $C^V(u_\tau)$ from 1 to $c^V(u_\tau)$, and use an iterator to indicate which record we are visiting. Geometric sampling works as follows: assume the iterator currently points to i (initially i=0), it generates a random number j from geometric distribution $Geom(1/d_\tau(u_\tau))$, then it *(i)* stops if $i+j > c^V(u_\tau)$; or *(ii)* jumps to $i+j$ and selects the pointed record. It is proved that this approach can sample each record with probability $1/d_\tau(u_\tau)$, and since the iterator has only visited the records which will be selected, the expected time for sampling will be $O(1)$. Nonetheless, it is hard to apply the approach to C^E efficiently. To explain, if we only maintain the auxiliary structure C^E on edges but not C^V on nodes, we need to find all random walks that pass through each edge $e \in \mathcal{E}_{\tau\text{-}1}(u_\tau)$ and then apply the geometric sampling approach to select the random walks to be adjusted. However, by examining each out-going edge of u, it already takes $O(d_\tau(u_\tau))$ time. We may make use of advanced data structures (e.g. segment-tree) to accelerate the procedure of seeking for the selected random walk among C^E. However, searching on a tree structure makes a factor of $O(\log d_\tau(u_\tau))$ increment of time complexity. More importantly, our motivation to use one auxiliary structure to support the tracing for both edge insertion and deletion is to reduce the space consumption of the auxiliary structure but the size of such an auxiliary structure of auxiliary structure to handle the whole graph will also be $O(m)$.

Our Solution. To tackle this issue, we first change the above geometric sampling approach to a sampling approach via the binomial distribution. In particular, we sample the number of random walk records (crossing u) from binomial distribution $B(c^V(u_\tau), 1/d_\tau(u_\tau))$, and then sample the records in $C^V(u_\tau)$ without replacement. It is easy to verify that this approach is equivalent to geometric sampling. However, the binomial sampling approach is also hard to apply to C^E directly to meet both time and space efficiency for similar reason as the geometric sampling approach.

To further overcome the obstacle, we present the following idea. In the process of binomial sampling, we repeat to sample a record from $C^V(u_\tau)$ without replacement. To avoid an iteration among $\mathcal{E}_{\tau\text{-}1}(u_\tau)$ (which takes $O(d(u_\tau))$ time), we can first sample an edge $e \in \mathcal{E}_{\tau\text{-}1}(u_\tau)$ which has at least one record (called an active edge) and then uniformly select one of the records $c \in C^E(e)$. It is obvious that if we select the edge e with probability $c^E(e)/c^V(u_\tau)$, we reproduce the uniformly sampling in $C^V(u_\tau)$. However, a weighted random sampling within $O(1)$ time requires the help of advanced data structures (e.g. alias array) with total size $O(m)$. Again, we will not build any additional data structure which has a total size of $O(m)$ since it goes against our intention to reduce space overheads. Instead, we turn to sample an edge $e \in \mathcal{E}_{\tau\text{-}1}(u_\tau)$ with probability $1/d'(u_\tau)$, where $d'(u_\tau)$ is the number of active edges and this number can be easily maintained with a counter for each node, saving far more space than maintaining auxiliary structure C^V on nodes.

To check if the alternative binomial sampling also keeps the unbiasedness, it needs to be clarified that we will prove the unbiasedness of H_τ in Section 5.1 only based on the fraction of the number of selected records in $C^E(e)$. Thus, any records passing through the same edge e can be equivalent for providing unbiasedness. Hence, if we select a record $c \in C^E(e)$, uniformly selecting another record $c' \in C^E(e)$ instead will not affect the unbiasedness. Assume that $c^V(u_\tau)=w$ and $d'(u_\tau)=d$, consider the probability that the record passing through a specific edge e will be selected by the original binomial sampling:

$$\begin{aligned}\mathbb{P}[c \in C^E(e)] &= \sum_{k=1}^{w} \mathbb{P}[c \in C^E(e) | c^E(e) = k] \cdot \mathbb{P}[c^E(e) = k] \\ &= \sum_{k=1}^{w} \frac{k}{w} \cdot \binom{w}{k} \left(\frac{1}{d}\right)^k \left(1 - \frac{1}{d}\right)^{w-k}. \end{aligned} \tag{5}$$

Firstly, we have that

$$\frac{k}{w} \cdot \binom{w}{k} = \frac{k}{w} \cdot \frac{w!}{k! \cdot (w-k)!} = \frac{(w-1)!}{(k-1)! \cdot (w-k)!} = \binom{w-1}{k-1},$$

and thus Equation 5 will be

$$\begin{aligned}\mathbb{P}[c \in C^E(e)] &= \sum_{k=1}^{w} \binom{w-1}{k-1} \left(\frac{1}{d}\right)^k \left(1 - \frac{1}{d}\right)^{w-k} \\ &= \frac{1}{d} \cdot \sum_{k=0}^{w-1} \binom{w-1}{k} \left(\frac{1}{d}\right)^k \left(1 - \frac{1}{d}\right)^{w-1-k} = \frac{1}{d}, \end{aligned}$$

where the last equality is due to the fact that the cumulative term is the summation of the probability mass function of $B(w-1, 1/d)$ and thus equals to 1. Therefore, if $H_{\tau\text{-}1}$ satisfies the unbiasedness, for any fixed $c^V(u_\tau)$ and $d'(u_\tau)$, the original binomial sampling method samples a record from a particular edge e with probability $1/d'(u_\tau)$, which is consistent with the strategy which samples the edges of u_τ uniformly. Since the unbiasedness of $H_{\tau\text{-}1}$ implies that the number of records crossing u and the number of active out-going edges of u are also unbiased, we can see that our strategy is equivalent to the original binomial sampling to keep the unbiasedness of H_τ.

So far, we have the final solution to sample records among $C^E(e)$. Algorithm 4 shows the pseudo-code of the new sampling method, where we use $c(u)$ to denote the total number of records crossing u. Note that $c(u)$ is equal to $c^V(u)$ which is the size of $C^V(u)$ but we store $c(u)$ as a counter directly on each node, which takes $\Theta(n)$ space and is much smaller than $O(m)$ in practice. Now return to Algorithm 4, it first generates the number of records to be selected from the binomial distribution $B(c_\tau(u_\tau), 1/d_\tau(u_\tau))$ (Line 2). To sample each record, it uniformly samples an edge $e \in \mathcal{E}_{\tau\text{-}1}(u_\tau)$ which is active and then uniformly samples one record in $c^E(e)$ (Lines 4-7). Finally, it

returns the sampled records. We apply Algorithm 4 to Algorithm 2 instead of sampling directly in $C^V(u_\tau)$ (Line 2) to select the affected random walks. For other parts, e.g., Lines 3-7 in Algorithm 3, the sampling approaches are unchanged.

5 THEORETICAL ANALYSIS

5.1 Proofs of Correctness

Proof of Theorem 4.3. Firstly, Algorithm 2 samples in $C^V_{\tau\text{-}1}(u_\tau)$ and updates the sampled random walks independently. In addition, for the newly sampled random walks (Lines 9-10), they are sampled independently and are independent from other random walks in H_τ. Thus, H_τ satisfies the independence property. Then, since the only difference between G_τ and $G_{\tau\text{-}1}$ is the edge $e_\tau = \langle u_\tau, v_\tau \rangle$, we add additional random walks starting from u_τ to satisfy the adequateness (Lines 9-10).

Next, we focus on unbiasedness. Clearly, the additional random walks are unbiased. Thus we will omit the clarification of them below. Consider the random walks in $H_{\tau\text{-}1}$ which never cross u_τ. Let $\pi(s,t;\bar{u})$ denote the contribution to $\pi(s,t)$ without crossing node u. Therefore,

$$\pi_\tau(s,t;\bar{u}_\tau) = \pi_{\tau\text{-}1}(s,t;\bar{u}_\tau), \tag{6}$$

because the insertion of e_τ makes no effect on the random walks which never cross u_τ. Since $H_{\tau\text{-}1}$ is unbiased and Algorithm 2 makes no change to the random walks which never cross u_τ in $H(s)$ (denoted as $H(s;\bar{u})$, correspondingly, the crossing subset denoted as $H(s;u)$), we obtain that,

$$\mathbb{E}\left[\frac{H_\tau(s,t;\bar{u}_\tau)}{H_\tau(s)}\right] = \pi_\tau(s,t;\bar{u}_\tau). \tag{7}$$

For the other part of random walks that still cross u_τ, let $\pi(s,t;e)$ denote the contribution to $\pi(s,t)$ with a passing through of edge $e = \langle u_\tau, v \rangle$. To simplify the discussion, for the present, we assume any random walk will not cross u_τ more than once and will discuss how to deal with the case that crosses u multiple times later. Due to the memorylessness of the random walk process, we have

$$\begin{aligned}\pi_\tau(s,t;e) &= \mathbb{P}[w \in T_\tau(e|s)] \cdot \pi_\tau(v,t;\bar{u}_\tau) \\ &= \frac{1}{d_\tau(u_\tau)} \cdot \mathbb{P}[w \in T_\tau(u_\tau|s)] \cdot \pi_\tau(v,t;\bar{u}_\tau),\end{aligned} \tag{8}$$

where $w \in T(e|s)$ indicates the s-starting random walk w passes through edge e and $w \in T(u|s)$ indicates such a random walk crosses node u. Notably,

$$\mathbb{P}[w \in T_\tau(u_\tau|s)] = \mathbb{P}[w \in T_{\tau\text{-}1}(u_\tau|s)] \tag{9}$$

since a random walk must cross u_τ before decides to pass through e_τ or not. Combining Equations 6,8,9 and since $d_\tau(u_\tau) = d_{\tau\text{-}1}(u_\tau) + 1$ with the insertion of e_τ, for $e = \langle u_\tau, v \rangle \in \mathcal{E}_{\tau\text{-}1}(u_\tau)$, we have

$$\begin{aligned}\pi_\tau(s,t;e) &= \frac{d_{\tau\text{-}1}(u_\tau)}{d_\tau(u_\tau)} \cdot \frac{1}{d_{\tau\text{-}1}(u_\tau)} \cdot \mathbb{P}[w \in T_{\tau\text{-}1}(u_\tau|s)] \cdot \pi_{\tau\text{-}1}(v,t;\bar{u}_\tau) \\ &= \left(1 - \frac{1}{d_\tau(u_\tau)}\right) \cdot \pi_{\tau\text{-}1}(s,t;e),\end{aligned} \tag{10}$$

Lastly and clearly, the contribution of the inserted edge e_τ is

$$\begin{aligned}\pi_\tau(s,t;e_\tau) &= \frac{1}{d_\tau(u_\tau)} \cdot \mathbb{P}[w \in T_\tau(u_\tau|s)] \cdot \pi_\tau(v,t;\bar{u}_\tau) \\ &= \frac{1}{d_\tau(u_\tau)} \cdot \mathbb{P}[w \in T_{\tau\text{-}1}(u_\tau|s)] \cdot \pi_\tau(v,t;\bar{u}_\tau).\end{aligned} \tag{11}$$

In Algorithm 2, we sample each u_τ-crossing record with probability $1/d_\tau(u_\tau)$ thus making the $1/d_\tau(u_\tau)$ fraction of descent to the contribution of random walks passing through $e \in \mathcal{E}_{\tau\text{-}1}(u_\tau)$ in expectation which keeps pace with Equation 10. Then we force the sampled random walks to cross the edge e_τ and continue randomly walking on G_τ which exactly matches Equation 11. Due to the

unbiasedness of $H_{\tau\text{-}1}$, the sum among $\mathcal{E}_\tau(u_\tau)$ leads to

$$\mathbb{E}\left[\frac{H_\tau(s,t;u_\tau)}{H_\tau(s)}\right] = \pi_\tau(s,t;u_\tau). \tag{12}$$

Equations 7 and 12 imply that H_τ is unbiased if a random walk will not cross node u_τ more than once. To show that H_τ is still unbiased even if a random walk can cross u_τ multi-times, let us consider a particular random walk in H_τ. Algorithm 2 independently samples each of u_τ-crossing records with probability $d_\tau(u)$. Once a record c is sampled, the rest path of the random walk $c.id$ after step $c.step$ will be discarded. Thus if multiple records of the same random walk are sampled, only the earliest one in the crossing order takes effect finally. Since the first time i a random walk will turn to e_τ after it crosses u_τ obey a geometric distribution $Geom(1/d_\tau(u_\tau))$ which consists of the probability that the i-th crossing record of it will be the dominant item in Algorithm 2, we finish the proof. □

Proof of Theorem 4.6. Algorithm 3 first trims $H_{\tau\text{-}1}$ by uniformly sampling random walks from $H_{\tau\text{-}1}(u_\tau)$. Let $H'_{\tau\text{-}1}$ denote the trimmed $H_{\tau\text{-}1}$, and the independence and unbiasedness of $H'_{\tau\text{-}1}$ will be kept the same as $H_{\tau\text{-}1}$ for the property of uniform sampling. Then, the trimming stops when the size of $H'_{\tau\text{-}1}$ satisfies the adequateness exactly. Next, we repair the invalid random walks which pass through the recently deleted edge e_τ. According to Equation 7 and since we make no change to the random walks without crossing u_τ, we have that Equation 6 also holds for the deletion case.

For the other part of random walks, the deletion of e_τ makes the contribution of e_τ become 0 and leads to an increment in the contribution of other edges $e \in \mathcal{E}_\tau(u_\tau)$. By Equations 6,8,9 and since $d_\tau(u_\tau)$=$d_{\tau\text{-}1}(u_\tau)$−1 with the deletion of e_τ, we can see that

$$\begin{aligned}\pi_\tau(s,t;e) &= \left(1+\frac{1}{d_\tau(u_\tau)}\right)\cdot\frac{1}{d_{\tau\text{-}1}(u_\tau)}\cdot\mathbb{P}[w\in T_\tau(u_\tau|s)]\cdot\pi_\tau(v,t;\bar{u}_\tau)\\ &= \pi_{\tau\text{-}1}(s,t;e)+\frac{1}{d_{\tau\text{-}1}(u_\tau)}\cdot\pi_\tau(s,t;e),\end{aligned}$$

holds with the assumption that any random walk will not cross node u_τ more than once. In Algorithm 3, we repair a random walk passing through e_τ by restarting it at u_τ, thus,

$$\begin{aligned}\mathbb{E}\left[\frac{{}_\Delta H_\tau(s,t;e)}{H_\tau(s)}\right] &= \mathbb{E}\left[\frac{H_{\tau\text{-}1}(s;e_\tau)}{H_{\tau\text{-}1}(s)}\right]\cdot\frac{1}{d_\tau(u_\tau)}\cdot\pi_\tau(v,t;\bar{u}_\tau)\\ &= \frac{1}{d_{\tau\text{-}1}(u_\tau)d_\tau(u_\tau)}\cdot\mathbb{P}[w\in T_{\tau\text{-}1}(u_\tau|s)]\cdot\pi_\tau(v,t;\bar{u}_\tau)\\ &= \frac{1}{d_{\tau\text{-}1}(u_\tau)}\cdot\pi_\tau(s,t;e),\end{aligned}$$

where ${}_\Delta H_\tau(s,t;e)$ denotes $H_\tau(s,t;e)$−$H_{\tau\text{-}1}(s,t;e)$ which is the increment of $H(s,t;e)$ at timestamp τ, the second equality is based on the unbiasedness of $H_{\tau\text{-}1}$ and the third equality comes from Equations 8-9. As the result of the above equations, we have

$$\mathbb{E}\left[\frac{H_\tau(s,t;e)}{H_\tau(s)}\right] = \pi_\tau(s,t;e). \tag{13}$$

Thus, Equations 7 and 13 imply that H_τ is unbiased if a random walk will not cross u_τ more than once. Finally, the correctness of Algorithm 3 comes from the fact that the deletion of e_τ makes no effect on a random walk unless it passes through e_τ, and the rest part of such a random walk behind the first time it passes through e_τ is entirely stale in the deletion case. □

5.2 Proofs of Efficiency

Proof of Theorem 4.4. As Algorithm 2 shows, the cost of **Update-Insert** contains two parts. One is caused by the modification of random walks whose records are sampled in set $C^V_{\tau\text{-}1}(u_\tau)$ to reflect

the change of out-neighbors of u_τ due to the arrival of edge e_τ. Let c_τ denote the number of random walks, we have

$$\mathbb{E}[c_\tau] \leq \sum_{u \in V_{\tau\text{-}1}} \mathbb{E}\left[|C^V_{\tau\text{-}1}(u)|\right] \cdot \frac{1}{d_\tau(u)} \cdot \mathbb{P}\left[e_\tau \in \mathcal{E}_\tau(u)\right],$$

where the inequality is because we will update a random walk at most once even if multiple of its records are sampled. Then with Equation 1, it yields that

$$\begin{aligned}\mathbb{E}[c_\tau] &\leq \sum_{u \in V_{\tau\text{-}1}} \mathbb{E}\left[|C^V_{\tau\text{-}1}(u)|\right] \cdot \frac{1}{d_\tau(u)} \cdot \frac{d_\tau(u)}{m_\tau} \\ &= \frac{1}{m_\tau} \cdot \sum_{u \in V_{\tau\text{-}1}} \mathbb{E}\left[|C^V_{\tau\text{-}1}(u)|\right].\end{aligned}$$

In addition, according to Lemma 3.2, the number of random walks starting from node s should be $\lceil d(s) \cdot r_{max} \cdot \omega \rceil$. Let $h_{\tau\text{-}1}(s,u)$ be the expected times that an s-starting random walk crosses node u at timestamp τ-1. It holds that

$$h_{\tau\text{-}1}(s,u) = \frac{1-\alpha}{\alpha} \cdot \pi_{\tau\text{-}1}(s,u), \tag{14}$$

where $\pi_{\tau\text{-}1}(s,u)$ is the PPR score of node u with respect to s on graph $G_{\tau\text{-}1}$. To explain, $\pi_{\tau\text{-}1}(s,u)$ can be written as follows:

$$\pi_{\tau\text{-}1}(s,u) = \sum_{i=0}^{\infty} \pi^i_{\tau\text{-}1}(s,u),$$

where $\pi^i_{\tau\text{-}1}(s,u)$ is the probability that an s-starting random walk stops at u exactly at the i-th hop. Therefore, the probability that an s-starting random walk visits node u at the i-th hop can be written as $\pi^i_{\tau\text{-}1}(s,u)/\alpha$. Multiplying it by $1-\alpha$, we derive the probability that an s-starting random walk crosses u at the i-th hop is $\pi^i_{\tau\text{-}1}(s,u) \cdot (1-\alpha)/\alpha$. Summing all hops together, we derive that $h_{\tau\text{-}1}(s,u)$, the expected times an s-starting random walk crossing u, satisfies Equation 14. Then, we could express $|C^V_{\tau\text{-}1}(u)|$ as follows:

$$\mathbb{E}\left[|C^V_{\tau\text{-}1}(u)\right] = \sum_{s \in V_{\tau\text{-}1}} \lceil d_{\tau\text{-}1}(s) \cdot r_{max} \cdot \omega \rceil \cdot h_{\tau\text{-}1}(s,u). \tag{15}$$

Let $V^*_{\tau\text{-}1}$ denote the set of nodes with at least one out-going edge, and let $n^*_{\tau\text{-}1}$=$|V^*_{\tau\text{-}1}|$. Combining the above equations, we obtain

$$\begin{aligned}\mathbb{E}[c_\tau] &\leq \frac{1}{m_\tau} \cdot \sum_{u \in V_{\tau\text{-}1}} \sum_{s \in V^*_{\tau\text{-}1}} \lceil d_{\tau\text{-}1}(s) \cdot r_{max} \cdot \omega \rceil \cdot \frac{1-\alpha}{\alpha} \cdot \pi_{\tau\text{-}1}(s,u) \\ &\leq \frac{1-\alpha}{\alpha} \cdot \frac{r_{max} \cdot \omega}{m_\tau} \cdot \sum_{s \in V^*_{\tau\text{-}1}} d_{\tau\text{-}1}(s) \sum_{u \in V_{\tau\text{-}1}} \pi_{\tau\text{-}1}(s,u) \\ &\quad + \frac{1-\alpha}{\alpha} \cdot \frac{1}{m_\tau} \cdot \sum_{s \in V^*_{\tau\text{-}1}} \sum_{u \in V_{\tau\text{-}1}} \pi_{\tau\text{-}1}(s,u)\end{aligned}$$

where the second equality results from that $d_{\tau\text{-}1}(s)$ is positive and thus $\lceil d_{\tau\text{-}1}(s) \cdot r_{max} \cdot \omega \rceil \leq d_{\tau\text{-}1}(s) \cdot r_{max} \cdot \omega + 1$. Due to the fact that $\sum_{u \in V_{\tau\text{-}1}} \pi_{\tau\text{-}1}(s,u) = 1$, we have

$$\begin{aligned}\mathbb{E}[c_\tau] &\leq \frac{1-\alpha}{\alpha} \cdot \frac{r_{max} \cdot \omega}{m_\tau} \cdot \sum_{s \in V_{\tau\text{-}1}} d_{\tau\text{-}1}(s) + \frac{1-\alpha}{\alpha} \cdot \frac{n^*_{\tau\text{-}1}}{m_\tau} \\ &< \frac{1-\alpha}{\alpha} \cdot (r_{max} \cdot \omega + 1) = O(r_{max} \cdot \omega),\end{aligned}$$

where the second inequality comes from $\sum_{s \in V_{\tau\text{-}1}} d_{\tau\text{-}1}(s) = m_{\tau\text{-}1} = m_\tau - 1$, and $n^*_{\tau\text{-}1} \leq m_{\tau\text{-}1}$ due to the definition of $n^*_{\tau\text{-}1}$.

Another part of cost is to expand $H_\tau(u_\tau)$ by adding more random walks starting from u_τ into H_τ to satisfy the adequateness, we have

$$\begin{aligned}|H_\tau(u_\tau)| - |H_{\tau\text{-}1}(u_\tau)| &= \lceil d_\tau(u_\tau) \cdot r_{max} \cdot \omega \rceil - \lceil d_{\tau\text{-}1}(u_\tau) \cdot r_{max} \cdot \omega \rceil \\ &= O\left(r_{max} \cdot \omega\right)\end{aligned}$$

holds since $d_\tau(u_\tau)$=$d_{\tau\text{-}1}(u_\tau)$+1. Finally, by setting $r_{max}\cdot\omega$=$\Theta(1)$ according to SpeedPPR [44], we achieve the result $\mathbb{E}[c_\tau]$=$O(1)$ and $|H_\tau(u_\tau)|$−$|H_{\tau\text{-}1}(u_\tau)|$=$O(1)$, that is, we need to update only $O(1)$ random walks, and add only $O(1)$ new random walks.

In Section 4.3, we presented an efficient sampling technique that takes only $O(1)$ to sample a random walk to update. Thus the cost of the procedure **Sample** (Line 2) is $O(1)$, due to $\mathbb{E}[c_\tau]$=$O(1)$. When updating a random walk, we need to invoke procedure **Walk-Restart** (Line 6). Since the expected length of a random walk with a decay factor α is $O\left(1/\alpha\right)$ =$O(1)$, the cost of **Walk-Restart** is $O(1)$. Similarly, the cost of adding a new random walk is also $O(1)$. From the above discussions, we know that **Update-Insert** takes $O(1)$ expected time to update the index for each insertion. □

Proof of Theorem 4.7. As Algorithm 3 shows, the cost of **Update-Delete** also contains two parts. First, we need to trim $H_{\tau\text{-}1}$ to $H'_{\tau\text{-}1}$ by sampling random walks from $H_{\tau\text{-}1}(u_\tau)$ uniformly. Since the trimming process of $H_\tau(u_\tau)$ in the deletion case is a reverse process of the expanding process of $H_\tau(u_\tau)$ in the insertion case, we have the cost $|H_{\tau\text{-}1}(u_\tau)|$−$|H_\tau(u_\tau)|$=$|H_{\tau\text{-}1}(u_\tau)|$−$|H'_{\tau\text{-}1}(u_\tau)|$=$O(r_{max}\cdot\omega)$.

Then, a deletion of edge e_τ= $\langle u, v\rangle$ leads to the rebooting of all random walks (except the trimmed random walks) that have records in $C^E_{\tau\text{-}1}(e_\tau)$. Let c_τ denote the number of random walks that need to be repaired, the expectation of c_τ is

$$\mathbb{E}\left[c_\tau\right] \leq \sum_{e\in E_{\tau\text{-}1}} \mathbb{E}\left[\left|C^E_{\tau\text{-}1}(e)\right|\right] \cdot \mathbb{P}\left[e_\tau = e\right],$$

where the inequality is due to the fact that we only need to update a random walk once even if it passes through e_τ multi-times and the records in $C^E_{\tau\text{-}1}$ that belong to trimmed random walks are also staled. Then, for each edge e= $\langle u, v\rangle \in E_{\tau\text{-}1}$, as a corollary of the definition of random walk, it follows that

$$\mathbb{E}\left[\left|C^E_{\tau\text{-}1}(e)\right|\right] = \frac{1}{d_{\tau\text{-}1}(u_\tau)} \cdot \mathbb{E}\left[\left|C^V_{\tau\text{-}1}(u)\right|\right].$$

Combining the equations above and Equation 2, we obtain

$$\begin{aligned}\mathbb{E}[c_\tau] &\leq \sum_{u\in V_{\tau\text{-}1}} \sum_{e\in \mathcal{E}_{\tau\text{-}1}(u)} \mathbb{E}\left[\left|C^E_{\tau\text{-}1}(e)\right|\right] \cdot \frac{1}{m_{\tau\text{-}1}} \\ &= \frac{1}{m_{\tau\text{-}1}} \cdot \sum_{u\in V_{\tau\text{-}1}} \sum_{e\in \mathcal{E}_{\tau\text{-}1}(u)} \frac{1}{d_{\tau\text{-}1}(u_\tau)} \cdot \mathbb{E}\left[\left|C^V_{\tau\text{-}1}(u)\right|\right] \\ &= \frac{1}{m_{\tau\text{-}1}} \cdot \sum_{u\in V_{\tau\text{-}1}} \mathbb{E}\left[\left|C^V_{\tau\text{-}1}(u)\right|\right].\end{aligned}$$

Given the above inequality, we can follow the same analysis steps as the insertion case, i.e., Equations 14-15 and the analysis after Equation 15 in the proof of Theorem 4.4, to derive that $\mathbb{E}[c_\tau]$=$O(r_{max}\cdot\omega)$. As $r_{max}\cdot\omega$=$\Theta(1)$ following SpeedPPR [44], it yields that both $|H_{\tau\text{-}1}(u_\tau)|$−$|H_\tau(u_\tau)|$=$O(1)$ and $\mathbb{E}[c_\tau]$=$O(1)$. Furthermore, as we described in the proof of Theorem 4.4, the cost of maintaining the index when updating or deleting a random walk can be bounded by $O(1)$. Hence, we get that **Update-Delete** takes expected $O(1)$ time to update the index. □

6 OTHER RELATED WORK

Personalized PageRank (PPR) was first proposed by Page et al. [32]. They present a matrix-based definition of PPR and explore the Power-Iteration method. Given a source s, let $\boldsymbol{\pi}(s)$ denote the PPR vector of PPR scores of each node with respect to s. Let $\boldsymbol{A}$ be the adjacency matrix and $\boldsymbol{D}$ be the diagonal matrix where the (u, u)-th entry is the out-degree of u. The following equation holds:

$$\boldsymbol{\pi}(\boldsymbol{s}) = \alpha \cdot \boldsymbol{e}_s + (1 - \alpha) \cdot \boldsymbol{\pi}(\boldsymbol{s}) \cdot \boldsymbol{D}^{-1}\boldsymbol{A},$$

where $\boldsymbol{e}_s$ is a one-hot vector with only the s-th entry to be 1. The Power-Iteration method is still expensive and this motivates a series of works to solve the above linear system more efficiently [14, 22, 28, 35, 49] via matrix related tricks, e.g., matrix decomposition. However, such methods are shown to be dominated by the Forward-Push + Monte-Carlo based methods as shown in [37, 43].

Another line of solutions are local push methods. The Forward-Push [6] algorithm can be used to derive the answer of single-source PPR query. However, it provides no guarantee on the answers. The Backward-Push [5, 21] is further proposed to derive the single-target PPR (STPPR) queries. Ohsaka et al. [30] and Zhang et al. [47] further design algorithms to update the stored Forward-Push results on dynamic graphs. In [47], Zhang et al. further present algorithms to maintain the stored backward push results. Wang et al. [37] further present randomized Backward-Push to gain a better trade-off between query efficiency and accuracy. However, all these methods cannot be applied to answer ASSPPR/ASSPPR top-k queries.

To gain an approximation guarantee, Monte-Carlo methods [11] are proposed to derive approximate estimations. However, Monte-Carlo methods alone are still too slow, which motivates existing solutions to combine local push algorithms and Monte-Carlo methods to gain better query efficiency while still providing approximation guarantee. In particular, Lofgren et al. [25, 26] and Wang et al. [39, 40] present solutions to combine random walk and Backward-Push to improve the query performance of pairwise PPR queries with approximation guarantees. Later, FORA [41], ResAcc [23], and SpeedPPR [44] are further proposed to combine the Forward-Push and random walks to improve the query performance for ASSPPR queries. The index-based version of FORA and SpeedPPR, dubbed as FORA+ and SpeedPPR+, respectively, are shown to incur high update costs as in our experiment. Besides, ResAcc is an index-free method while its query processing is not as fast as such index-based solutions.

There also exist a line of research works, e.g., [13, 14, 41, 43] on efficient top-k PPR query processing. The state-of-the-art approach is FORA+, which achieves the best query efficiency as shown in [41], but it is an index-based method. Previously there exist no efficient algorithms to support dynamic index update and our FIRM fills this gap. The state-of-the-art index-free method is TopPPR [43], which combines Forward-Push, random walk, and Backward-Push to answer top-k PPR queries with precision guarantees. However, TopPPR is only designed for top-k queries and cannot support SSPPR queries. In contrast, with the same random walk index, our FIRM supports both efficient ASSPPR and ASSPPR top-k queries.

Finally, there exist research works on parallelizing PPR computations with multi-core [38], GPU [16, 34], or in distributed environment [15, 20, 23, 27, 33]. These works are orthogonal to ours.

7 EXPERIMENTS

Next, we experimentally evaluate our FIRM against alternatives. All experiments are conducted on an AWS x1.16xlarge cloud server with 64vCPUs clocked at 2.3GHz and 976GB memory. Source codes [4] used in experiments are all implemented in C++ and compiled with full optimization.

Abbr.	Name	n	m	Type
SF	*Stanford*	281.9K	2.3M	directed
DB	*DBLP*	317.1K	1.0M	undirected
YT	*Youtube*	1.1M	3.0M	undirected
PK	*Pokec*	1.6M	30.6M	directed
LJ	*LiveJournal*	4.8M	69.0M	directed
OK	*Orkut*	3.1M	117.2M	undirected
TW	*Twitter*	41.7M	1.5B	directed
FS	*Friendster*	65.6M	1.8B	undirected

Table 5. Datasets. ($K = 10^3, M = 10^6, B = 10^9$)

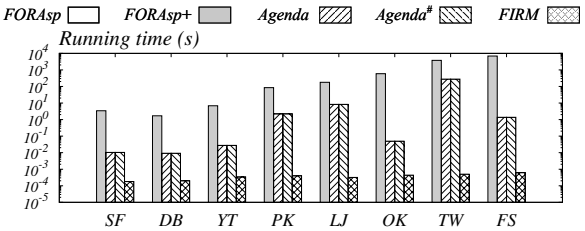


Fig. 2. Average Update Time

7.1 Experimental Settings

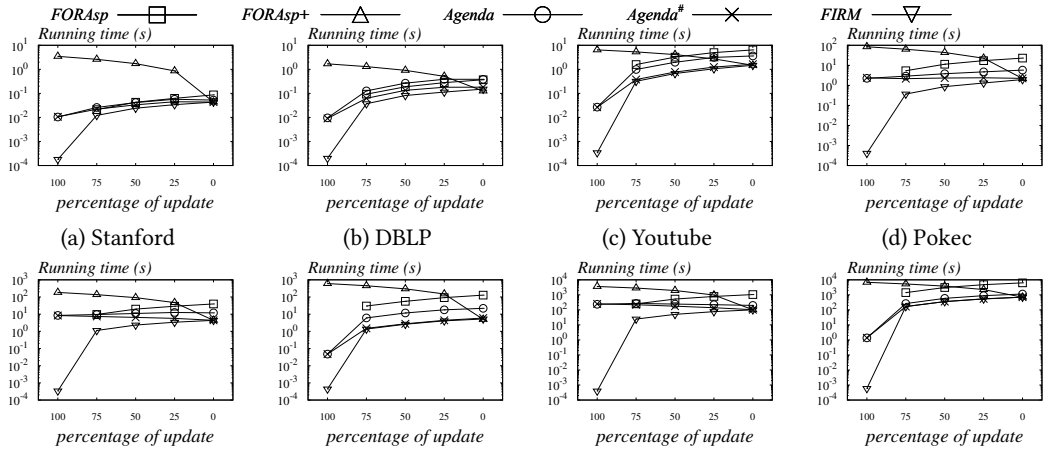
We compare our method against four solutions. FORAsp is the method whose number of random walks is set to $O(m)$ following SpeedPPR [44] whereas its workflow is identical to the original FORA [41] because Power-Push of SpeedPPR on evolving graphs is not as efficient as that on static graphs. FORAsp+ is the index-based version of FORAsp. The state-of-the-art solution for evolving graphs, Agenda [29], is also included. We further include Agenda$^{\#}$, a variant of Agenda, which has been discussed in Section 3.2.

Datasets and Metrics. We use 8 benchmark datasets that can be obtained from public sources SNAP[2] and Konect[1] and are frequently used in previous research works on PPR, e.g., [25, 29, 41, 44], as shown in Table 5. To measure the performance of the solutions on evolving graphs, for each dataset, we randomly shuffle the order of edges and divide it into two parts. The first part which has 90% edges (50% edges for Twitter and Friendster to reduce the running time on cloud servers) will be used to build the initial graph. Then, we generate workloads each consisting of 100 updates/queries. An update will be either *(i)* an insertion of an edge selected randomly from the rest part of the edges, or *(ii)* a deletion of an edge selected randomly from the initial graph. A workload with update percentage x% means that it contains x updates and $(100-x)$ queries.

Parameter Settings. Following previous work [41, 44], we set α=0.2, ϵ=0.5, δ=$1/n$ and p_f=$1/n$ by default. In addition, we set Agenda according to [29] such that θ=0.5 and r^b_{max}=$d_\tau(u_\tau)/m_\tau$ on undirected graphs or r^b_{max}=$1/n_\tau$ on directed graphs. For top-k queries, we set k=500. To speed up query processing of index-based FORA-like methods, we balance the time cost between Forward-Push and refining phases by setting $r_{max}\cdot\omega$=β/α, where β is a parameter depending on the dataset.

7.2 Performance of FIRM

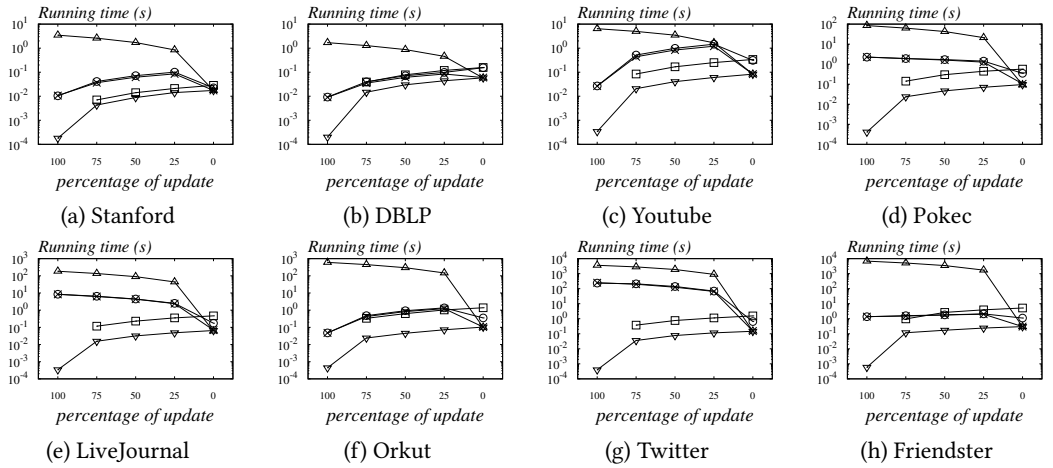
Update Performance. We evaluate the efficiency of FIRM against index-based alternatives for updating the index structure. Figure 2 shows the average processing time for each update under



(a) Stanford (b) DBLP (c) Youtube (d) Pokec

(e) LiveJournal (f) Orkut (g) Twitter (h) Friendster

Fig. 4. Average processing time with ASSPPR queries



(a) Stanford (b) DBLP (c) Youtube (d) Pokec

(e) LiveJournal (f) Orkut (g) Twitter (h) Friendster

Fig. 5. Average processing time with ASSPPR top-k queries.

a workload with 50% update. Observe that FORAsp+ has the worst update performance since it simply rebuilds its index which leads to a prohibitive computation. Compared to FORAsp+, Agenda does improve the update performance. Note that, the update process of Agenda is just to trace the inaccuracy of its index, the affected random walks will be reconstructed during query processing. Agenda$^{\#}$ has the same update process as Agenda. Our solution, FIRM, is orders of magnitude faster than FORAsp+ and Agenda. Moreover, FIRM has a similar time consumption among all datasets which confirms that our solution takes $O(1)$ time to maintain its index for each update. In contrast, the update time of FORAsp+ and Agenda/Agenda$^{\#}$ increases notably with graphs becoming larger.

General Performance. We first reveal the general performance of FIRM, Figure 4 shows the performance under different workloads consisting of edge updates and ASSPPR queries. In a word, our solution FIRM outperforms all alternatives under an arbitrarily mixed workload for all the datasets. The advantage of FIRM is more prominent for workloads that consist of edge updates and ASSPPR top-k queries as shown in Figure 5. This is because the query cost of a top-k query is much slighter than a full ASSPPR query so the update cost will significantly affect the performance.

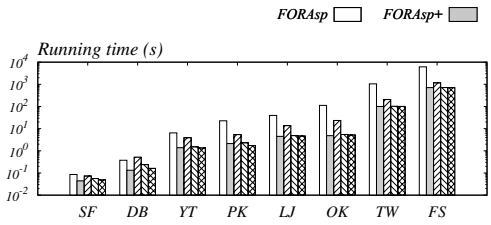


Fig. 6. Average Query Time of ASSPPR Queries

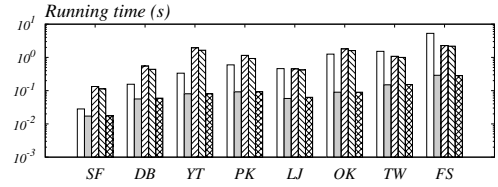


Fig. 7. Average Query Time of ASSPPR Top-k Queries

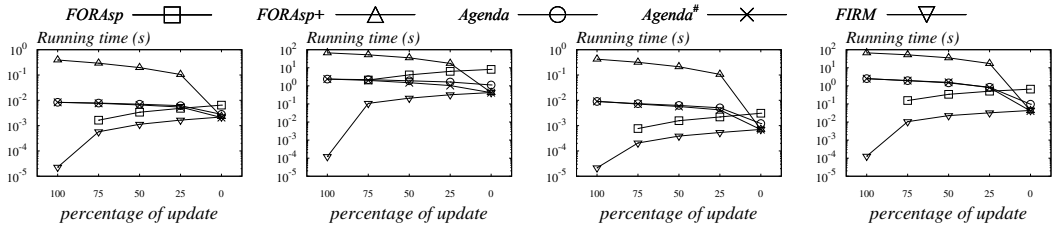
Therefore, FIRM has great practical value since ASSPPR top-k query is widely used for web-search, recommendation systems, and other scenarios. Besides, the essence of an algorithm for top-k queries is to provide a rough but adequate precision of SSPPR. Thus, FIRM can also be applied to the scenarios which need a loose (ϵ, δ)-approximation guarantee (e.g. δ=$O(1)$) on evolving graphs to improve their performance.

Full Query Performance. We compare the average time for ASSPPR query processing under a workload with an update percentage of 50 to reflect the additional cost of the lazy-update strategy of Agenda/Agenda#. Figure 6 shows the average query time of ASSPPR queries. Not surprisingly, FORAsp is the slowest one to answer ASSPPR queries because it is an index-free approach. FIRM is as fast as FORAsp+ for query processing and achieves over 10x speed-up over FORAsp on most datasets. Agenda is faster than FORAsp but slower than FORAsp+ and FIRM because it may have to reconstruct some random walks before query processing, and need more computation in Forward-Push phase to provide the same approximation guarantee since it admits inaccuracy tolerance when updating the index. Agenda# takes slightly more running time than FORAsp+ and FIRM, as it needs to apply the lazy-update scheme during query processing. However, with the avoidance of additional computation in Forward-Push phase, it is surely faster than Agenda.

Top-k Query Performance. The situation is not quite the same when processing ASSPPR top-k queries. As Figure 7 shows, Agenda is even slower than FORAsp on some datasets. To explain, the process to answer top-k queries in [41] repeats invoking Forward-Push with a rough r'_{max}>r_{max}, refining the temporary result to provide a (ϵ, δ')-approximation guarantee where δ'>δ is a rough threshold and checking whether δ' is enough to bound the top-k PPR scores. After each time Forward-Push is invoked, Agenda must check and fix the inaccuracy of its index, making the performance analysis in [29] not applicable anymore.

Agenda# suffers from the same problem as well. Besides, it has to be mentioned that in Figure 5 the query performance of Agenda# (with 100% query) is almost as fast as FORAsp+ and FIRM, while in Figure 7 its performance is much worse. It seems paradoxical at first glance. To explain, it is a tricky optimization for a workload with a low update rate. We can easily maintain an upper bound for the total error of the current index, and then if the upper bound is below error tolerance, we are allowed to skip the lazy-update phase. Thus, the lazy-update process is never invoked under a pure query workload. However, under mixing workloads, the lazy-update process is frequently invoked, thus becoming the bottleneck of Agenda#. In contrast, our solution still keeps the same query performance as FORAsp+ for top-k queries and achieves over 10x speed-up over FORAsp.

Performance on Real-World Temporal Graphs. To examine the effectiveness of FIRM in real-world scenarios, we have run FIRM and alternatives on two temporal social networks, Digg (0.28M nodes and 1.7M edges) and Flickr (2.3M nodes and 33.1M edges), where edge timestamps (i.e., when the edge is built) are provided. The two graphs can also be downloaded at [1]. To simulate the evolving process, we design the experiment as follows: We first sort all edges by the edge



(a) Digg (b) Flickr (c) Digg Top-k (d) Flickr Top-k

Fig. 7. Average processing time on temporal graphs.

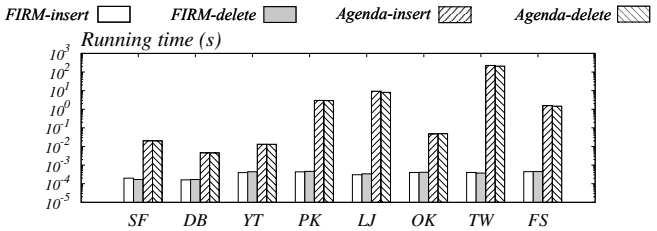


Fig. 8. Average cost of insertion and deletion

timestamp in ascending order and take the first 90% edges as the initial graph; then remaining edges will be sequentially added to the graph according to their timestamps. Figure 7 shows the average processing time of our FIRM and its competitors for the ASSPPR queries. From the figures, we can see that FIRM still keeps superb efficiency on real-world evolving graphs with orders of magnitude speedup. We further examine the update cost of FIRM and Agenda on these two graphs which are generated under the random arrival model rather than sorted by their actual timestamps. The results are in our technical report [3], which shows that FIRM (as well as Agenda) has similar update performance under these two edge arrival settings, with negligible gap.

Performance of Insertion and Deletion. To compare the updating cost of edge insertion with that of edge deletion, we respectively run 1000 insertions and 1000 deletions on each dataset, by using FIRM and Agenda. We omit FORAsp+ because its update scheme is the same for edge insertion and deletion. Agenda$^{\#}$ is also omitted because it takes the same updating cost as Agenda as shown in Figure 2. Figure 8 shows the average running time of edge insertion and deletion. As we can observe, with our FIRM, the update cost for edge insertion is almost equal to that for edge deletion, which confirms our theoretical analysis that both of them have $O(1)$ cost. Agenda also has a similar trend, consistent with the analysis in [29].

Accuracy. To evaluate the accuracy performance of FIRM and its competitors, we first perform a sufficient number of updates (5∼10 percent edges are inserted) and then measure the relative error of ASSPPR queries. We stop the process if it cannot finish in 48 hours (thus we have no accuracy results for Agenda/Agenda$^{\#}$ on large graphs). As to FORAsp+, since its accuracy performance is independent of the update process (it always reconstructs the whole index for each update), we can save computational cost by constructing the index only once, after all updates have been applied. The experimental results are shown in Figure 10, where the box (resp. bar) represents the average (resp. maximum) relative error. FIRM has the identical precision as FORAsp/FORAsp+, which verifies the correctness of our solution. Besides, as discussed in Section 3.2, Agenda has a practical precision higher than other methods which comes from its conservative bound of index inaccuracy and hence a tighter bound in FORA phase. Correspondingly, Agenda$^{\#}$ gives a precision comparable to (slightly worse than) FORAsp/FORAsp+. Even though we make an aggressive assumption on the inaccuracy of its lazy-updated index, which means the precision of Agenda$^{\#}$ will be theoretically worse than

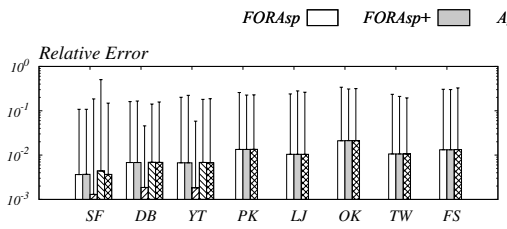


Fig. 10. Accuracy results

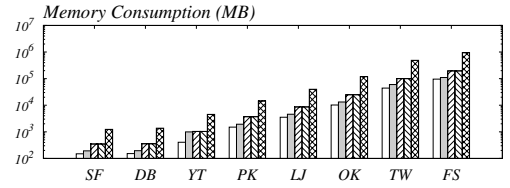


Fig. 11. Memory Consumption

FORAsp/FORAsp+, the gap of average relative error between Agenda$^{\#}$ and FORAsp/FORAsp+ is not significant. As we described in Sections 4 and 3.2, $O(1)$ changes of random walks are adequate to guarantee the precision. However, the lazy-update scheme renews more random walks since it traces the inaccuracy roughly. Therefore, in most cases, the inaccuracy of the lazy-updated index is not as large as Agenda supposed. In summary, our FIRM achieves the best query and update efficiency when providing identical accuracy as alternatives including Agenda, where we tune it as Agenda$^{\#}$ to gain similar accuracy as FIRM. In our technical report [3], we have also done experiments with only 1000 updates so that Agenda/Agenda$^{\#}$ can finish in a reasonable time. We have similar observations as discussed above.

Memory Consumption. Figure 11 shows the memory consumption of all methods. FORAsp needs to maintain the evolving graph itself. Then, FORAsp+ costs about 2x memory of FORAsp to store the terminals of pre-sampled random walks, and Agenda/Agenda$^{\#}$ requires more additional space to store the reverse graph to support Backward-Push. FIRM costs about 8x space as that of FORAsp+ to trace the random walks efficiently. However, with the new sampling scheme introduced in Section 4.3, FIRM can still handle huge-scale graphs like Twitter and Friendster in memory. In practice, several times more space consumption is an acceptable trade-off for orders of magnitude times speed-up of processing performance because expanding the memory is much easier than enhancing computing power. Moreover, there are several techniques, e.g. $\sqrt{1-\alpha}$ random walk [43], to make full use of the complete path for more accurate estimation and thus can reduce the space without loss of precision.

8 CONCLUSIONS

In this paper, we present FIRM, an efficient framework to handle approximate single source PPR problems on evolving graphs. Theoretical analysis proves that our proposal has $O(1)$ time cost for each update in expectation and experiments show that FIRM dramatically outperforms competitors in most scenarios.

ACKNOWLEDGMENTS

Sibo Wang is supported by the Hong Kong RGC ECS grant (No. 24203419), Hong Kong RGC GRF grant (No. 14217322), Hong Kong RGC CRF grant (No. C4158-20G), Hong Kong ITC ITF grant (No. MRP/071/20X), NSFC grant (No. U1936205), and a gift from Huawei. Zhewei Wei is supported in part by the major key project of PCL (PCL2021A12), by the National Natural Science Foundation of China (No. 61972401, No. 61932001), by Beijing Natural Science Foundation (No. 4222028), by Peng Cheng Laboratory, by Alibaba Group through Alibaba Innovative Research Program, by the CCF-Baidu Open Fund (NO. 2021PP15002000) and by the Huawei-Renmin University joint program on Information Retrieval.

REFERENCES

[1] 2013. KONECT. http://konect.cc/networks/.
[2] 2014. SNAP Datasets. http://snap.stanford.edu/data.
[3] 2022. Technical Report. https://arxiv.org/abs/2212.10288.
[4] 2023. Source Code. https://github.com/lalumine/firm.
[5] Reid Andersen, Christian Borgs, Jennifer T. Chayes, John E. Hopcroft, Vahab S. Mirrokni, and Shang-Hua Teng. 2007. Local Computation of PageRank Contributions. In *WAW*. 150–165.
[6] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. 475–486.
[7] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast Incremental and Personalized PageRank. *Proc. VLDB Endow.* 4, 3 (2010), 173–184.
[8] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling Graph Neural Networks with Approximate PageRank. In *SIGKDD*. 2464–2473.
[9] Fan Chung and Linyuan Lu. 2006. Concentration Inequalities and Martingale Inequalities: A Survey. *Internet Mathematics* 3, 1 (2006), 79–127.
[10] Xinyu Du, Xingyi Zhang, Sibo Wang, and Zengfeng Huang. 2023. Efficient Tree-SVD for Subset Node Embedding over Large Dynamic Graphs. *PACMMOD* 1, 1 (2023), 96:1–96:26.
[11] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358.
[12] Dongqi Fu and Jingrui He. 2021. SDG: A Simplified and Dynamic Graph Neural Network. In *SIGIR*. 2273–2277.
[13] Yasuhiro Fujiwara, Makoto Nakatsuji, Hiroaki Shiokawa, Takeshi Mishima, and Makoto Onizuka. 2013. Efficient ad-hoc search for personalized PageRank. In *SIGMOD*. 445–456.
[14] Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka. 2012. Efficient personalized pagerank with accuracy assurance. In *KDD*. 15–23.
[15] Tao Guo, Xin Cao, Gao Cong, Jiaheng Lu, and Xuemin Lin. 2017. Distributed Algorithms on Exact Personalized PageRank. In *SIGMOD*. 479–494.
[16] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel Personalized Pagerank on Dynamic Graphs. *PVLDB* 11, 1 (2017), 93–106.
[17] Xingzhi Guo, Baojian Zhou, and Steven Skiena. 2021. Subset Node Representation Learning over Large Dynamic Graphs. In *KDD*. ACM, 516–526.
[18] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: The who to follow service at twitter. In *WWW*. 505–514.
[19] Zoltán Gyöngyi, Pavel Berkhin, Hector Garcia-Molina, and Jan O. Pedersen. 2006. Link Spam Detection Based on Mass Estimation. In *VLDB*. 439–450.
[20] Guanhao Hou, Xingguang Chen, Sibo Wang, and Zhewei Wei. 2021. Massively Parallel Algorithms for Personalized PageRank. *PVLDB* 14, 9 (2021), 1668–1680.
[21] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *WWW*. 271–279.
[22] Jinhong Jung, Namyong Park, Lee Sael, and U Kang. 2017. BePI: Fast and Memory-Efficient Method for Billion-Scale Random Walk with Restart. In *SIGMOD*. 789–804.
[23] Dandan Lin, Raymond Chi-Wing Wong, Min Xie, and Victor Junqiu Wei. 2020. Index-Free Approach with Theoretical Guarantee for Efficient Random Walk with Restart Query. In *ICDE*. 913–924.
[24] Wenqing Lin. 2019. Distributed Algorithms for Fully Personalized PageRank on Large Graphs. In *WWW*. 1084–1094.
[25] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. 2015. Bidirectional PageRank Estimation: From Average-Case to Worst-Case. In *WAW 2015*. 164–176.
[26] Peter Lofgren, Siddhartha Banerjee, Ashish Goel, and Comandur Seshadhri. 2014. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *KDD*. 1436–1445.
[27] Siqiang Luo. 2019. Distributed PageRank Computation: An Improved Theoretical Study. In *AAAI*. 4496–4503.
[28] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken ichi Kawarabayashi. 2014. Computing personalized PageRank quickly by exploiting graph structures. *PVLDB* 7, 12 (2014), 1023–1034.
[29] Dingheng Mo and Siqiang Luo. 2021. Agenda: Robust Personalized PageRanks in Evolving Graphs. In *CIKM*. 1315–1324.
[30] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. 2015. Efficient PageRank Tracking in Evolving Networks. In *SIGKDD*. 875–884.
[31] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric Transitivity Preserving Graph Embedding. In *KDD*. 1105–1114.
[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: bringing order to the web.* Technical Report. Stanford InfoLab.

[33] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. 2013. Fast Distributed PageRank Computation. In *ICDCN*. 11–26.
[34] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. 2019. Realtime Top-k Personalized PageRank over Large Graphs on GPUs. *PVLDB* 13, 1 (2019), 15–28.
[35] Kijung Shin, Jinhong Jung, Lee Sael, and U. Kang. 2015. BEAR: Block Elimination Approach for Random Walk with Restart on Large Graphs. In *SIGMOD*. 1571–1585.
[36] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. 2018. VERSE: Versatile Graph Embeddings from Similarity Measures. In *WWW*. 539–548.
[37] Hanzhi Wang, Zhewei Wei, Junhao Gan, Sibo Wang, and Zengfeng Huang. 2020. Personalized PageRank to a Target Node, Revisited. In *SIGKDD*. 657–667.
[38] Runhui Wang, Sibo Wang, and Xiaofang Zhou. 2019. Parallelizing approximate single-source personalized PageRank queries on shared memory. *VLDB J.* 28, 6 (2019), 923–940.
[39] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. 2016. HubPPR: Effective Indexing for Approximate Personalized PageRank. *PVLDB* 10, 3 (2016), 205–216.
[40] Sibo Wang and Yufei Tao. 2018. Efficient Algorithms for Finding Approximate Heavy Hitters in Personalized PageRanks. In *SIGMOD*. 1113–1127.
[41] Sibo Wang, Renchi Yang, Runhui Wang, Xiaokui Xiao, Zhewei Wei, Wenqing Lin, Yin Yang, and Nan Tang. 2019. Efficient Algorithms for Approximate Single-Source Personalized PageRank Queries. *TODS* 44, 4 (2019), 18:1–18:37.
[42] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. In *SIGKDD*. 505–514.
[43] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Shuo Shang, and Ji-Rong Wen. 2018. TopPPR: Top-k Personalized PageRank Queries with Precision Guarantees on Large Graphs. In *SIGMOD*. 441–456.
[44] Hao Wu, Junhao Gan, Zhewei Wei, and Rui Zhang. 2021. Unifying the Global and Local Approaches: An Efficient Power Iteration with Forward Push. In *SIGMOD*. 1996–2008.
[45] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S. Bhowmick. 2020. Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank. *Proc. VLDB Endow.* 13, 5 (2020), 670–683.
[46] Yuan Yin and Zhewei Wei. 2019. Scalable Graph Embeddings via Sparse Transpose Proximities. In *KDD*. 1429–1437.
[47] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate Personalized PageRank on Dynamic Graphs. In *KDD*. 1315–1324.
[48] Xingyi Zhang, Kun Xie, Sibo Wang, and Zengfeng Huang. 2021. Learning Based Proximity Matrix Factorization for Node Embedding. In *KDD*. 2243–2253.
[49] Fanwei Zhu, Yuan Fang, Kevin Chen-Chuan Chang, and Jing Ying. 2013. Incremental and Accuracy-Aware Personalized PageRank through Scheduled Approximation. *PVLDB* 6, 6 (2013), 481–492.

Received April 2022; revised July 2022; accepted August 2022