One Index for All: Towards Efficient Personalized PageRank Computation for Every Damping Factor

JUNJIE ZHOU, Beijing Institute of Technology, China
MEIHAO LIAO, Beijing Institute of Technology, China
RONG-HUA LI, Beijing Institute of Technology, China
LONGLONG LIN, Southwest University, China
GUOREN WANG, Beijing Institute of Technology, China

Personalized PageRank (PPR) is a fundamental graph proximity measure with a wide range of applications. The single-source Personalized PageRank (SSPPR) query aims to compute the PPR values for all nodes from a given source node. Due to the high computational cost of exact SSPPR computation, most existing solutions focus on approximate queries with accuracy guarantees. The state-of-the-art approaches for approximate SSPPR queries are index-based and are typically designed for a single, fixed value of α. However, real-world applications often require handling multiple values of α, and current index-based methods struggle to adapt to varying values of α without rebuilding the index. To address this limitation, we propose a novel and efficient index approach for SSPPR queries that supports all values of α. A striking feature of our approach is that it stores only a single index for all possible values of α. This is achieved by leveraging the loop-erased α-random walk interpretation of PPR and constructing a stack-style *meta-index* with a sufficiently large damping factor $\bar{\alpha}$, denoted as StackIndex. Then, we develop a novel technique to efficiently transform StackIndex into a *new index* for any specified damping factor α, without the need to rebuild the index. We show that our index construction algorithm requires around $O(\omega n)$ time and $O(\omega n)$ space to ensure approximation quality, where ω and n represent the sample size and the number of nodes in the graph, respectively. We also develop an index maintenance technique to update our StackIndex when handling dynamic graphs with edge insertions and deletions. Extensive experiments on 5 large real-world graphs demonstrate that StackIndex offers substantial speedups over previous index-based methods for PPR queries with different α while maintaining the same accuracy guarantee.

CCS Concepts: • **Networks → Network algorithms**; • **Mathematics of computing → Probabilistic algorithms**.

Additional Key Words and Phrases: personalized pagerank; dynamic algorithm; index maintenance

1 Introduction

Given a directed graph $G = (V, E)$ and a damping factor α, Personalized PageRank (PPR) is a function that takes two vertices from V as input and outputs a pairwise proximity score [14, 16, 19, 32]. The PPR value $\pi(s, t)$ quantifies the relative importance of node t to node s. Informally, $\pi(s, t)$ is defined as the probability that an α-random walk, starting from s, terminates at t. An α-random walk is a process where, at each step, there is a probability α of terminating at the current node,

Authors' Contact Information: Junjie Zhou, zjjyyyk@bit.edu.cn, Beijing Institute of Technology, Beijing, China; Meihao Liao, mhliao@bit.edu.cn, Beijing Institute of Technology, Beijing, China; Rong-Hua Li, lironghuabit@126.com, Beijing Institute of Technology, Beijing, China; Longlong Lin, longlonglin@swu.edu.cn, Southwest University, Chongqing, China; Guoren Wang, wanggrbit@126.com, Beijing Institute of Technology, Beijing, China.

ACM 2836-6573/2025/9-ART258
https://doi.org/10.1145/3749176

and a probability $1 - \alpha$ of moving to a random out-neighbor of the current node. As a fundamental proximity measure, PPR is widely used in various applications, including web search [19, 32], graph clustering [2, 35], and graph neural network [7, 10, 21, 47]. This broad applicability has led to increasing challenges in designing efficient and effective PPR algorithms.

The computation of the single-source PPR vector, i.e., $\pi(s, t)$ for every $t \in V$, can be efficiently reformulated as solving a corresponding linear system, as demonstrated in [32]. However, traditional linear solvers generally exhibit a time complexity of $O(|V|^3)$, rendering them impractical for real-world applications. To mitigate this, numerous efficient algorithms have been proposed that leverage graph-specific properties. These methods can be broadly categorized into three main approaches [24, 26, 41, 44, 45]: Monte Carlo, power iteration, and local push. These techniques are often combined to enhance accuracy while reducing the overall time complexity. For instance, the state-of-the-art (SOTA) method employs a two-stage framework [45]. The first stage approximates the PPR using the local push method. The second stage then refines this approximation by leveraging a Monte Carlo sampling-based index. This two-stage method achieves high accuracy within a relative error framework [26, 41, 44, 45].

In real-world applications, however, the damping factor α varies depending on the specific task at hand. For instance, local graph clustering often requires a PPR with $\alpha \approx 0.01$ [2, 35], whereas PPR used as a graph proximity measure typically necessitates $\alpha \approx 0.2$ [26, 41]. Furthermore, some earlier studies have suggested that the α value in PPR should be tailored to individual users [13] and source nodes [4, 5]. Nevertheless, conventional two-stage PPR approximation algorithms are designed to accommodate only a single α value, as their indexes are constructed by recording information (such as random walk trajectories, end nodes, etc.) from α-random walks originating from each vertex i. Various α necessitates different indexes, and since these indexes often consume substantial storage space, storing multiple indexes can be prohibitively expensive.

To address this challenge, we propose an efficient and versatile indexing approach, termed StackIndex, which can handle different damping factors α in PPR computations using only one index. Our approach is based on the classic loop-erased random walk sampling algorithm introduced by Wilson [43], which was originally designed to sample uniform spanning trees in graphs. Such an algorithm was recently extended to compute the single-source PPR vector in [26], where a loop-erased random walk interpretation of the PPR vector is established. As shown by Wilson [43], such a loop-erased random walk sampling algorithm can be interpreted as a stack popping procedure. Specifically, the stack popping procedure maintains a stack S_j for each node j, with each stack element independently chosen from the out-neighbors of j. The selection of out-neighbors is performed randomly, in proportion to the out degree, which exactly simulates a random walk step. Based on such a stack popping procedure and the results established in [26], we develop StackIndex where each element in our StackIndex consists of two components: a Next vector and a Root vector. The Next vector corresponds to the spanning forest obtained by the stack popping procedure, representing the next node to visit in the random walk. The Root vector records the root of each node in the spanning forest, effectively tracking the connected components throughout the process. Both vectors are of size $O(n)$, where n denotes the number of nodes in the graph. Armed with these two vectors, we can efficiently approximate the PPR vector based on the results established in [26].

To support single-source PPR queries with different damping factors α, we initially construct a meta-index StackIndex using a variant of the Wilson algorithm with a sufficiently-large damping factor $\bar{\alpha}$. This meta-index is designed to handle a wide range of damping factors efficiently. We show that StackIndex can be constructed in $O(\omega n)$ time and requires $O(\omega n)$ space, where ω denotes the sample size. Subsequently, to answer a PPR query (s, α), we develop a novel technique to transform the meta-index into a *new index* tailored to the specific damping factor α. This *new* *index* is then used to approximate the PPR vector with the damping factor α. We rigorously prove that the approximated PPR computation using the *new index* is equivalent to using a freshly-built index with the damping factor α. This implies that our approach allows a single meta-index to handle PPR queries for any damping factor α without requiring the construction of an index for each α. Furthermore, we propose a new approach to dynamically update the meta-index when dealing with evolving graphs, without rebuilding the meta-index from scratch. In summary, the main contributions are as follows.

Novel Index Solution. We propose StackIndex, a novel meta-index that can handle PPR queries with different damping factors α using a single index, constructed in $O(\omega n)$ time and space. We develop a novel technique to transform the meta-index into a new index tailored to any damping factor α, enabling efficient PPR computation without rebuilding the index. We also present a detailed theoretical analysis of the correctness and complexity of our solution.

New Index Maintenance Algorithm. We propose a new and efficient meta-index updating algorithm for dynamic graphs that handles edge insertions and deletions. A key feature of our approach is its ability to skip unnecessary computations without requiring a complete index rebuild.

Extensive Experiments. We conducted extensive experiments on five large-scale real-world datasets to validate our solution. The results demonstrate that our approach significantly improves query speed over state-of-the-art algorithms while maintaining the same level of accuracy for PPR queries. Additionally, the experiments show that our index maintenance method can be up to two orders of magnitude faster than the update method based on rebuilding. Furthermore, we conducted two case studies highlighting how StackIndex efficiently facilitates the selection of the optimal α, enhancing performance in real-world applications.

Reproducibility. The source code of this paper: https://github.com/zjjyyyk/stackindex.

2 Preliminaries

Given a directed graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, a damping factor α, a source node s, and a target node t, personalized PageRank (PPR) $\pi(s,t)$ is defined as the probability that an α-random walk starting from s terminates at t. An α-random walk is a variant of the traditional random walk where, at each step, there is a probability α that the walk terminates, and a probability $1-\alpha$ that it jumps to a uniformly-selected neighbor of the current node. Let P denote the transition probability of random walk, PPR can also be stated in matrix form: $\pi_s = \alpha e_s + (1-\alpha)\pi_s P^T$, where e_s represents the standard n-dimensional vector with 1 in s-th coordinate and 0 elsewhere. In this paper, we focus on the following PPR query:

Definition 1 (Single-Source PPR Query [26, 41, 44, 45]). *Given a source node s and a damping factor α, a single-source* PPR *query $QUERY(s,\alpha)$ is to compute an approximation of the* PPR *vector $\tilde{\pi}(s,\cdot)$, where $\tilde{\pi}(s,t)$ approximates $\pi(s,t)$ for all $t \in V$.*

This problem becomes particularly challenging when the damping factor α is not fixed but varies across queries. Unlike previous studies [26, 41, 44], where α is often assumed to be constant, we consider a more realistic setting where α varies across queries. This variation is essential in practical applications where real-world systems must handle scenarios requiring flexibility for α as discussed in Section 1. Throughout this paper, we assume $\alpha \le \bar{\alpha}$ for all queries, where $\bar{\alpha}$ is a pre-specified upper bound that is typically observed to be around 0.3 in real-world systems [2, 24, 26, 35, 41, 44].

To evaluate the quality of the query results, we aim to design algorithms that satisfy rigorous error guarantees, which are critical for ensuring the reliability and accuracy of the computed PPR. We consider the following two types of error guarantees widely used in previous studies [26, 41, 44].

Table 1. Frequently used notations

Symbol	Description	Symbol	Description	Symbol	Description
π_s	single-source PPR from s	α, β	damping factor, $\beta = \frac{\alpha}{1-\alpha}$	$N(u)$	out-neighbor set
$d_{out}(u)$	out-degree	$\bar{\alpha}$	meta-damping factor	Δ_{out}	maximum out-degree
P	probability transition matrix	e_s	unit vector, $e_s[s] = 1$	$\lambda_i(P)$	i-th largest eigenvalue of P
$Tr(A)$	trace of matrix A	W	constant w.r.t. ϵ, δ, p_f	ω	sample size
τ_α	spanning forest sampling time w.r.t. α				

Algorithm 1: Forward-Push [2]

```
Input: Graph $G$, source node $s$, damping factor $\alpha$, threshold $r_{\max}$
Output: $\hat{\pi}(s,\cdot)$ and residual $r(s,\cdot)$
1 for each $u \in V, u \neq v$ do
2     $r(s,u) = 0, \hat{\pi}(s,u) = 0$;
3 $r(s,s) = 1$;
4 while $\exists u \in V$ such that $r[u] \geq r_{\max}$ do
5     $\hat{\pi}(s,u)$ += $\alpha \cdot r(s,u)$;
6     for each $w \in N(u)$ do
7         $r(s,w)$ += $(1-\alpha)r[u]/d_{out}(u)$;
8     $r(s,u) = 0$;
9 return $\hat{\pi}(s,\cdot), r(s,\cdot)$;
```

Definition 2 (Relative-Error Guarantee). *Given error parameters ϵ, δ, and p_f, we say that the approximation $\tilde{\pi}(s,\cdot)$ satisfies relative-error guarantee if, for all $t \in V$ where $\pi(s,t) > \delta$, the following holds with probability at least $1 - p_f$:*

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

Definition 3 ($L1$-Error Guarantee). *Given an error parameter ϵ, we say that the approximation $\tilde{\pi}(s,\cdot)$ satisfies $L1$-error guarantee if the following holds:*

$$\sum_{t \in V} |\tilde{\pi}(s,t) - \pi(s,t)| \leq \epsilon.$$

These error guarantees provide a robust framework for evaluating the performance of approximation PPR algorithms in handling dynamic damping factors. By addressing the significant challenges of varying α, our work aims to bridge the gap between theoretical advancements and practical applications, providing a scalable and accurate approach for PPR computation under various α scenarios.

3 Existing Methods and Their Defects

3.1 Online Two-Stage Methods

The state-of-the-art online algorithms for single-source PPR computation all adhere to a two-stage framework [26, 29, 41, 44], encompassing two distinct stages. In the first stage, PPR values are deterministically computed using the classical forward-push method [2], as described in Algorithm 1. Specificly, Forward-Push computes PageRank by dynamically redistributing residual weights. Initially, each node is assigned a residual weight. The algorithm processes nodes with the highest residuals, retaining a portion proportional to α as the final value and redistributing the remaining $(1 - \alpha)$ weight evenly to its neighbors. If a neighbor's residual exceeds a predefined threshold, it is added to the processing queue. The process continues until all residuals fall below the threshold. The key idea of the two-stage method relies on the following *push invariant*:

$$\pi(s,t) = \hat{\pi}(s,t) + \sum_{u \in V} r(s,u)\pi(u,t). \tag{1}$$

The invariant in Eq. (1) holds throughout the forward push process. Based on this, the first term is computed via forward push, and the second is estimated using Monte Carlo sampling [29]. FORA [41] first adopted this hybrid approach for single-source PPR, and SpeedPPR [44] further improved it by enhancing forward push with the power method.

In the second stage, Monte-Carlo estimators refine the estimates by approximating each $\pi(u, t)$. This two-stage process reduces the overall time complexity while maintaining relative-error guarantee. There are two main Monte-Carlo estimators for PPR. The first is based on α-random walks: $\pi(s, t)$ is represented as the probability of an α-random walk from s terminating at t. FORA [41] and SpeedPPR [44] employ α-random walks as the Monte Carlo estimator. Setting $W = \frac{(2+2\epsilon/3)\log(\frac{1}{p_f})}{\epsilon^2\delta}$, FORA samples $\lceil r(s, u) \cdot W \rceil$ α-random walks on each node u to guarantee relative-error. On scale-free graphs, the overall sampling size of FORA is $O(\frac{n\log n}{\epsilon})$ for relative-error guarantee, whereas SpeedPPR only requires $O(n\log n \cdot \log(1/\epsilon)$, improving from $1/\epsilon$ to $\log(1/\epsilon)$. The second estimator uses spanning forests [26]: $\pi(s, t)$ is represented as the probability that s is rooted at t in a uniform rooted α-spanning forest (Definition 4). The SpeedL algorithm proposed in [26] adopts spanning forests as the Monte Carlo estimator, which is shown to be robust with respective to the damping factor α. It samples $\lceil r_{max} \cdot W \rceil$ spanning forests in total to achieve a relative-error guarantee.

The major drawback of all these online methods is their inefficiency in answering single-source PPR queries with varying damping factors. The random walk-based method performs better for large α values, while the random spanning forest-based method is more suitable for small α values. None of these methods can efficiently handle queries for all values of α.

3.2 Index-based Methods

Another approach to efficiently answer PPR queries is the use of index-based solutions [26, 41, 44]. The core idea behind existing index-based solutions, such as FORA+ [41], SpeedPPR+ [44], and SpeedL+ [26], is that they maintain random walk or random spanning forest samples as an index. These indices are then utilized to accelerate the Monte-Carlo stage of online two-stage methods. Unlike online two-stage methods, which require real-time sampling of random walks or random spanning forests, these index-based methods are often significantly faster because the sampling is precomputed and stored in the index.

The main limitation of all existing index-based solutions is that they treat the damping factor α as a constant and cannot accommodate different α values without rebuilding the index. In other words, if we need to compute the PPR values with a different α, the existing index cannot be used, and a new index must be reconstructed. This requirement is clearly impractical when handling our problem, as we cannot maintain too many indices with various α values. Consequently, a natural question arises: Can we design an index-based method capable of handling all α values?

However, answering the above question is a very challenging task because random walk or random spanning forest samples are highly dependent on the parameter α. First, with various values of α, the sample sizes required to achieve the same relative-error guarantee differ significantly. Second, the distributions of the random walk or random spanning forest samples can be quite different for different α values. Therefore, maintaining a single index that can efficiently support PPR queries with various α values is very challenging. In the following section, we will propose a novel index-based solution, StackIndex, to overcome these challenges.

4 A Novel Stack Index: StackIndex

Our StackIndex is based on the random α-spanning forest explanation of PPR proposed in [26]. We first builds a meta-index with a sufficiently-large damping factor $\bar{\alpha}$, represented as StackIndex$_{\bar{\alpha}}$, by obtaining a random $\bar{\alpha}$-spanning forest with respect to (w.r.t.) $\bar{\alpha}$. Compared to [26], in order to handle different α, in addition to storing the root information of nodes, we also store the whole spanning forest, which only takes $O(n)$ additional space but is essential to adapt the index for different α. To answer a query QUERY(s, α), our query processing algorithm first adapts the meta-index from $\bar{\alpha}$ to α. However, the problem of generating an α-spanning forest from the stored random $\bar{\alpha}$-spanning forest is very challenging. We propose an algorithm that is much faster than rebuilding a new index with α from scratch for this purpose. Then, we answer the query by incorporating the most advanced Forward Push technique proposed in [44] and the adapted new index, represented as StackIndex$_\alpha$, following [26]. Below, we first describe the index structure of StackIndex. Then, we introduce the index-building algorithm in Section 4.1. For the query processing, we first introduce the algorithm for adapting α in Section 4.2, and then describe the whole query processing algorithm in Section 4.3.

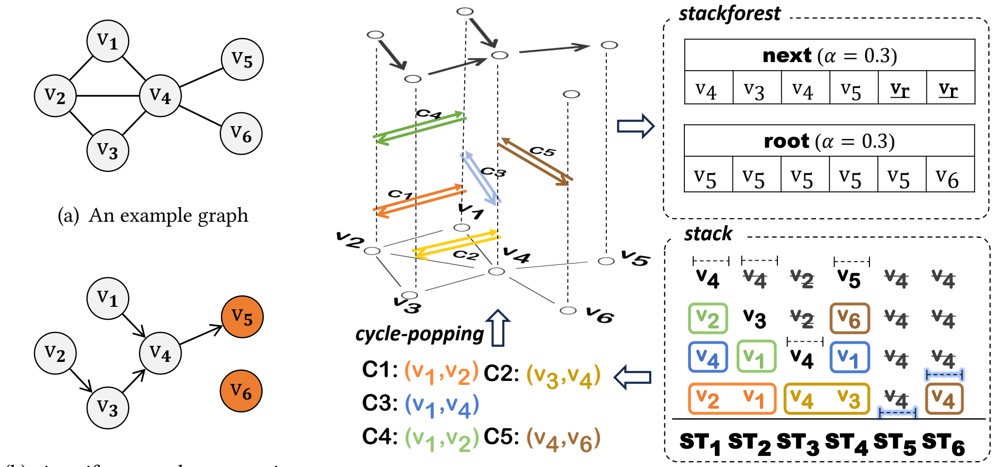
A tree in a graph is a connected acyclic graph where each node has an out-degree at most 1. A rooted tree designates one node as the root, with the root having an out-degree 0 and all other nodes having an out-degree 1. A rooted forest is a collection of rooted trees. A rooted spanning forest of a graph G is a subgraph containing all n nodes and forming a rooted forest. Based on these concepts, a uniform rooted α-spanning forest is defined as follows.

Definition 4 (Uniform Rooted α-Spanning Forest). *A rooted spanning forest F is uniform rooted α-spanning forest if it is sampled with probability proportional to the edge weights, multiplied by $\prod_{r\in\rho(F)} \beta d_{out}(r)$, where $\beta = \frac{\alpha}{1-\alpha}$ and $\rho(F)$ is the set of roots in F.*

Given a random walk with trajectory $w = (w_0, w_1, \cdots, w_n)$, the corresponding loop-erased random walk trajectory is defined as $L(w) = (l_0, l_1, \cdots, l_m)$ which satisfies: $l_0 = w_0$, $l_i = w_j$, where $j = \min\{k \mid k > i, w_k \notin \{l_0, l_1, \cdots, l_{i-1}\}$. Wilson's algorithm [43] utilizes loop-erased random walks to sample uniform rooted spanning trees. Specifically, given a graph G, we initiate a loop-erased random walk from each node v until all nodes have been visited. This process generates a uniform rooted spanning tree, where the parent of each node is its next node in the loop-erased random walk. In the algorithm, the loop-erased process can also be represented using a stack: each time the random walk visits a new node, the node is pushed onto the stack. When a node that is already in the stack is encountered, it indicates the formation of a cycle. In such cases, all nodes in the cycle are removed from the stack until the repeated node is reached. Extending Wilson's algorithm to spanning forests is straightforward: introduce a virtual node, connect it to all nodes in the graph, and execute Wilson's algorithm to obtain a spanning tree rooted at the virtual node. Finally, remove the virtual node to obtain the desired spanning forest.

Liao et al. [26] have established a connection between PPR and the probabilities of random spanning forests. Specifically, $\pi(s, t)$ can be represented as the probability that s is rooted in t in a random α-spanning forest. This is because, a uniform rooted α-spanning forest essentially unions several loop-erased walks with termination probability α, and by order-invariance property, each loop-erased walk can be viewed as an undisturbed random walk. This allows us to compute PPR unbiasedly following random walk interpretation of PPR while compressing time and memory consumption required to generate multiple independent random walks. Notably, as α decreases, the expected length of random walks increases, making compression effect of spanning forest-based approach more significant. The spanning forest-based approach has been shown to be robust and efficient for PPR computation [26].

Based on the results presented in [26], we can pre-compute the random spanning forests with respect to the meta damping factor $\bar{\alpha}$ to create a meta-index, denoted as StackIndex. This StackIndex is highly compact, as it only necessitates the storage of the sampled spanning forest corresponding to $\bar{\alpha}$. Specifically, each random spanning forest is represented by StackForest, which consists of two vectors of size n: the vector Next, which denotes the next node in the spanning forest for



(a) An example graph

(b) A uniform random spanning forest

(c) The index structure of StackIndex

Fig. 1. Illustration of the index structure of StackIndex

each node, and the vector Root, which indicates the root of each node in the spanning forest. For contrast, the index approach proposed in [26] only stores the Root. In our method, we additionally store the Next to capture the structural information of the spanning forest, enabling non-trivial adaptations of the $\bar{\alpha}$-spanning forest to different damping factors α.

Example 1. *Fig. 1(a)-(b) illustrates the index structure of the proposed* StackIndex. *The example graph in Fig. 1(a) serves as a demonstration. In Fig. 1(b), a uniform random spanning forest (*RSF*) is sampled using a Wilson-like algorithm with $\bar{\alpha}$. The spanning forest (Fig. 1(b)) is stored in the* Next *vector, where each element represents the out-neighbor of a node in the spanning forest. Additionally, the root of each node is recorded in the* Root *vector. For example, node v_1 points to v_4 and is rooted at v_5, so* Next$[v_1] = v_4$ *and* Root$[v_1] = v_5$. *Together, the* Next *and* Root *vectors form a* StackForest *in* StackIndex. *The construction process of* RSF *shown in Fig. 1(b) is detailed in Fig. 1(c), which will be further explained in Example 2 of Section 4.1.*

It is easy to derive that the index size of StackIndex is $O(\omega \cdot n)$, where ω is the sample size. Unlike traditional index-based methods, StackIndex is designed to handle different α values, making its sample size ω distinct. Following the reasoning in Section 3, to balance the time consumption across different stages, r_{max} should be set to $\sqrt{\frac{1}{\alpha W}}$. Thus, sample size should be $\min(\sqrt{\frac{W}{\alpha}}, W)$. We set the sample size ω to $\lceil W \rceil$ in StackIndex to accommodate all possible α values. It is worth noting that when $r_{max} < 1$, it is unnecessary to scan all samples; instead, scanning only $\lceil r_{max}\omega \rceil$ samples suffices. Importantly, the space consumption of StackIndex is independent of α, similar to other methods like the random walk-based index SpeedPPR+ and the random spanning forest-based index SpeedL+. However, a key advantage of StackIndex is its ability to handle queries with varying α values efficiently.

4.1 The Index-building Algorithm

To simulate the random spanning forest with the meta damping factor $\bar{\alpha}$, [26] utilizes a special type of random walk known as the loop-erased α-random walk to sample uniform rooted α-spanning forest efficiently. The loop-erased random walk is a random walk that removes cycles as they are formed. A uniform spanning forest is generated by fixing the loop-erased random walk trajectory and updating the boundary when all nodes are included in this acyclic subgraph [26]. To establish

Algorithm 2: The StackIndex Construction Algorithm

```
Input: Graph $G$, meta damping factor $\bar{\alpha}$, sample size $\omega$
Output: meta-index StackIndex$_{\bar{\alpha}}$
1  for $i = 1 : \omega$ do
2      InTree[$u$] ← False for $u \in V$;
3      for $u \in V$ do
4          c ← u;
5          while InTree[$c$] = False do
6              if $rand() \leq \alpha$ then
7                  Next [c] ← $v_r$;
8                  break;
9              Next [c] ← random out-neighbor of $c$; Stack.push(Next [c]);
10             $c$ ← Next[$c$];
11         if InTree[$c$] = False then
12             $r$ ← Root[$c$];
13         else
14             $r \leftarrow u$
15         $c \leftarrow u$;
16         while $c \neq u$ and InTree[$c$] = False do
17             InTree[$c$] ← True;
18             Root[$c$] ← $r$;
19             $c$ ← Next[$c$];
20     StackForest$_i$ ← (Next, Root);
21 return StackIndex$_{\bar{\alpha}}$ I = (StackForest$_i$)$_{i=1,2,\cdots,\omega}$;
```

the StackIndex, we also use this approach which is inspired by the Wilson algorithm [43] for generating uniform rooted spanning trees.

The algorithm uses two auxiliary vectors Next and InTree during the sample process. When the algorithm terminates, a uniform rooted α-spanning forest is stored in Next. Besides, the resulting StackIndex **I** stores ω such samples, where ω is the sample size.

Algorithm 2 outlines the construction process of the StackIndex. The algorithm initializes the necessary data structures, including Next, Root, InTree, and the stack Stack for each vertex u (line 2). Then, it performs a random walk from every vertex u. The algorithm continues the walk until this part of an existing forest or a random number in the range $[0, 1]$ is no more than α. It records the random walk trajectory to Stack (lines 5-10). Once the walk reaches an existing forest or terminates with probability α, the root node for the current tree is determined (lines 12-15), and the algorithm retraces the trajectory to update the InTree and Root (lines 16-19). After processing all vertices, the data structures are packed into StackForest$_i$ (line 20). This process is repeated ω iterations, yielding the final stack index **I**, which is returned as the StackIndex (line 21).

Example 2. *An example of* StackIndex$_{\bar{\alpha}}$ *is illustrated in Fig. 1(c), The algorithm starts by sampling random out-neighbors for each node, forming a spanning subgraph. For example, v_5 samples v_r as its out-neighbor and becomes part of the* InTree. *Cycles $C1 = (v_1, v_2)$ and $C2 = (v_3, v_4)$ appear but are eliminated by removing the corresponding edges and resampling the out-neighbors. This leads to a new cycle $C3 = (v_1, v_4)$. After eliminating $C3$ and resampling again, two more cycles, $C4 = (v_1, v_2)$ and $C5 = (v_4, v_6)$, emerge. Following the same process, v_6 samples v_r and becomes* InTree, *while the remaining nodes connect to v_5, converting to* InTree *as v_5 is* InTree. *Once all nodes are* InTree, *the algorithm terminates.*

We first prove that the index returned by Algorithm 2 can produce a uniform random α-spanning forest according to the desired distribution so that it can be used for PPR estimation. Due to space limits, the missing proofs in this paper can be found in Appendix A.1.

LEMMA 4.1. *(Correctness of Algorithm 2) Each* StackForest *of* StackIndex *returned by Algorithm 2 is a uniform rooted α-spanning forest of G.*

Below, we analyze the time and space complexity of Algorithm 2.

THEOREM 4.2. *(Time and space complexity of Algorithm 2) Algorithm 2 requires $O(\omega \cdot \mathrm{Tr}(I-\frac{P}{\beta+1})^{-1})$ time and $O(\omega \cdot (\mathrm{Tr}(I-\frac{P}{\beta+1})^{-1}+n))$ space complexity, where $\beta=\frac{\alpha}{1-\alpha}$.*

As observed in experiments (see Section 6.2, Figure 6), $\mathrm{Tr}(I-\frac{P}{\beta+1})^{-1}$ is approximately $O(n)$ in real-world graphs. As a result, the practical time complexity can be simplified to $O(\omega \cdot n)$.

Discussions. Notice that the time complexity of Algorithm 2 is strictly smaller than that of sampling α-random walks. For comparison, constructing the α-random walk-based index requires $O(\omega \cdot \frac{n}{\alpha})$ time. Since the eigenvalues of the transition matrix P lie in $[-1, 1]$, we have

$$\mathrm{Tr}\left(I-\frac{P}{\beta+1}\right)^{-1}=\sum_{i=0}^{n}\frac{1}{1-\frac{\lambda_i(P)}{\beta+1}}\leq\sum_{i=0}^{n}\frac{1}{1-\frac{1}{\beta+1}}=\frac{n}{\alpha}.$$

This demonstrates that the StackIndex algorithm is more efficient than the random walk-based algorithm in index building, particularly when the eigenvalues of the transition matrix P are significantly less than 1. Experiments in Section 6 also verifies this analysis (see Table 3).

Comparison with [26]. A key limitation of the approach in [26] is its inability to support PPR queries for varying α values without rebuilding the index from scratch. To address this, we introduce StackIndex, which incorporates an additional auxiliary structure, *next*, enabling efficient PPR queries across all α values. Compared to [26], the key innovations of StackIndex are summarized as follows: (i) StackIndex supports PPR queries for any α by incrementally adapting the meta-index, whereas [26] requires rebuilding the index for each new α; (ii) StackIndex can handle dynamic graph updates, including edge insertions and deletions, while [26] only addresses static graphs; (iii) We provide a comprehensive theoretical analysis of StackIndex, ensuring ϵ-error guarantees for the query process through carefully designed parameter settings.

4.2 Adapting StackIndex to Different α

When a query QUERY(s, α) is issued, as described earlier, our StackIndex can address this query by first adapting the index from $\bar{\alpha}$ to α. In this subsection, we propose an algorithm for adapting α from $\bar{\alpha}$ to the queried α. A notable feature of our method is its ability to adapt the index in significantly less time than would be required to rebuild the index from scratch. Specifically, we continue loop-erased α-random walk based on meta-index. The key idea lies in partially resampling the root set to achieve local adapting, and then the process continues by extending the loop-erased walk until a new StackForest is established.

At a high level, our algorithm proceeds as follows: (i) **Sample from the set $\rho_{\bar{\alpha}}$ to obtain ρ'_{α}.** Let $\rho_{\bar{\alpha}}$ denote the root set of the spanning forest sampled in the meta-index StackIndex$_{\bar{\alpha}}$. We sample a subset of nodes from $\rho_{\bar{\alpha}}$ and continue the sampling process with respect to the new root set ρ'_{α}; (ii) **Extend the loop-erased α-random walk from the new root set ρ'_{α}.** After sampling, we resample the out-neighbors of nodes in ρ'_{α} as their new outgoing edges. These new edges, combined with the existing edges in Next, may form additional cycles. By iteratively eliminating these cycles, we obtain a new spanning forest. As we will prove, this process results in a uniform rooted α-spanning forest.

Sample from the root set $\rho_{\bar{\alpha}}$. First, we sample a proportion of roots from the root set obtained in StackForest$_{\bar{\alpha}}$. The proportion p, however, needs to be determined. We start by noting that $\pi_{\alpha}(u, u)$

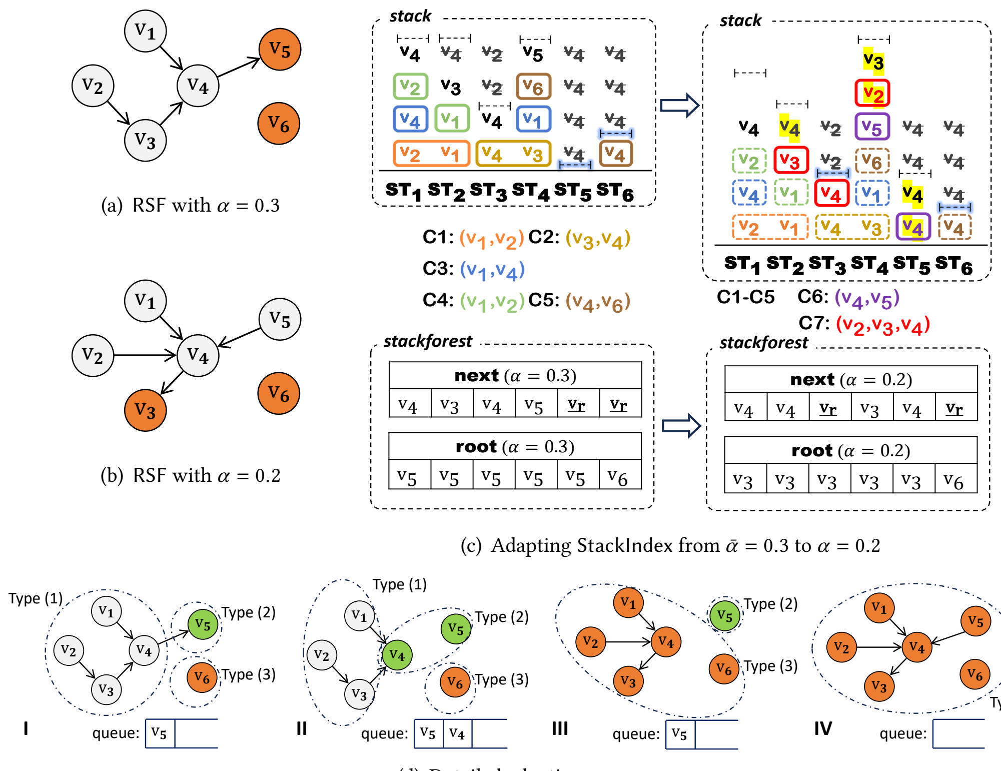

(a) RSF with $\alpha = 0.3$
(b) RSF with $\alpha = 0.2$
(c) Adapting StackIndex from $\bar{\alpha} = 0.3$ to $\alpha = 0.2$
(d) Detailed adapting process
stack
stackforest
next ($\alpha = 0.3$)
root ($\alpha = 0.3$)
next ($\alpha = 0.2$)
root ($\alpha = 0.2$)
C1: (v_1,v_2) C2: (v_3,v_4)
C3: (v_1,v_4)
C4: (v_1,v_2) C5: (v_4,v_6)
C1-C5 C6: (v_4,v_5)
C7: (v_2,v_3,v_4)
Type (1)
Type (2)
Type (3)
queue:

Fig. 2. An illustrative example of adapting StackIndex from $\bar{\alpha} = 0.3$ to $\alpha = 0.2$

is the probability of u being a root in a uniform rooted α-spanning forest. This probability can be decomposed into two cases: (1) $u \in \rho_{\bar{\alpha}}$ but not sampled, with probability $\pi_{\bar{\alpha}}(u,u) \cdot (1-p)$, and (2) u attends in additional loop-erased α-random walks and become a root. To determine the probability of case (2), note that the additional passes at u correspond to the difference in contributions when constructing indices with $\bar{\alpha}$ and α. Since $\pi_\alpha(u,u)$ can also be described as $\alpha \cdot \#\{\text{loop-erased } \alpha\text{-random walks from } u \text{ passes } u\}$ [26], the expected additional past times of u is $\Delta_u = \frac{\pi_\alpha(u,u)}{\alpha} - \frac{\pi_{\bar{\alpha}}(u,u)}{\bar{\alpha}}$. The probability for case (2) is then $\alpha \cdot \Delta_u = \pi_\alpha(u,u) - \frac{\alpha}{\bar{\alpha}} \cdot \pi_{\bar{\alpha}}(u,u)$. From this, we derive $p = 1 - \frac{\alpha}{\bar{\alpha}}$. Our theoretical analysis will confirm that this method produces a uniform α-spanning forest conditioned on the $\bar{\alpha}$-spanning forest.

Extend the loop-erased α-random walk from the sampled root set ρ'_α. After a set of nodes ρ'_α are sampled from $\rho_{\bar{\alpha}}$, we first resample a uniformly distributed out-neighbor (not including v_r) for each node in ρ'_α, then we continue loop-erased α-random walks from ρ'_α, conditioned on the StackForest$_{\bar{\alpha}}$. Specifically, during the process of the continued walk, when the node is first visited, we directly use the out-edge in StackForest$_{\bar{\alpha}}$. When a node is visited more than once, we sample a uniformly distributed out-neighbor of that node (including v_r) to continue the walk. The same construction process as the index building algorithm continues until there is no cycle on top of the stacks.

Algorithm details. To implement the idea of continuing loop-erased α-random walk, we use a *queue* to maintain nodes that still need to be visited from, and we classify the nodes into the following three types:

(1) Nodes that walking along Next eventually leads to Type (2).
(2) Nodes that Next values are to resample (or to say, in *queue*).
(3) Nodes that rooted at roots in $\rho_{\bar{\alpha}} \backslash \rho'_{\alpha}$.

Our goal is to convert all Type (1) and Type (2) nodes into Type (3) nodes. The distinction between Type (1) and Type (2) nodes is that Type (2) nodes require a new random sampling of a neighbor to determine their Next value, whereas Type (1) nodes already have a valid Next value. If no cycle exists along Next, Type (2) nodes will eventually be converted to Type (1) nodes. Consequently, converting all Type (2) nodes to Type (3) nodes ensures that Type (1) nodes naturally become Type (3) nodes as well.

After sampling ρ'_{α} from $\rho_{\bar{\alpha}}$, we initialize *queue* to ρ'_{α}. For each node in *queue*, we start by resampling its neighbor. Except for the first resampling of nodes in ρ'_{α}, the node becomes a root with probability α, converting it into a Type (3) node. Otherwise, we walk along Next starting from the node in the *queue*, stopping under one of the following conditions:

(1) The traversal reaches a Type (3) node, indicating that the nodes along the trajectory is rooted in a new spanning forest.
(2) The traversal reaches another Type (2) node (all nodes in the queue are Type (2) nodes), in which case the node transitions to Type (1), requiring no further operations.
(3) The traversal returns to the starting node, forming a cycle. In this case, all previously traversed nodes are converted to Type (2) nodes and added to the *queue*.

When *queue* is empty, all Type (2) nodes are converted to Type (3), so as the Type (1) nodes. Thus, Next forms a new rooted spanning forest again. We will prove that it is a uniform rooted α-spanning forest, which converts to $\mathsf{StackForest}_{\alpha}$. Notice that after locally transforming Next, there may still be nodes that were not traversed but whose Root information needs updating. For these nodes, we simply traverse the whole nodes set and update their Root values.

The pseudo-code for adapting StackIndex to a smaller α is shown in Algorithm 3. To adapt the index from the meta $\bar{\alpha}$ to a smaller $\alpha \leq \bar{\alpha}$, the algorithm processes each index element by updating its tree structure. For each element in the index $\mathsf{StackIndex}_{\bar{\alpha}}$, the root set of the corresponding StackForest is sampled with a proportion of $1-\frac{\alpha}{\bar{\alpha}}$ and recorded in the resampling *queue* Q (lines 1-4). Each node's *Type* is updated accordingly (line 5). The algorithm then iterates continuous random walks from nodes in Q. For each sampled root in Q, resample its neighbors (lines 6-11) and continue the random walk along Next until one of the following conditions is met: (i) it returns to the starting node, (ii) it reaches another node in Q, or (iii) it encounters a node u with $\mathsf{InTree}[u] = \mathsf{True}$. In the first case, all past nodes are pushed to Q and their *Type* is set to 1 (lines 14-15). In the third case, the Root of all past nodes is updated to $\mathsf{Root}[c]$, and their *Type* is set to 3 (lines 16-17). Finally we iterate over vertices and update their Root accordingly (lines 19-24). After processing all elements, the new index $\mathsf{StackForest}_{\alpha}$ is returned (line 26).

Example 3. *An example of our α-adapting approach is illustrated in Fig. 2. Fig. 2(a) and Fig. 2(b) show the random $\bar{\alpha}$-spanning forest and α-spanning forest, respectively, representing an example sample of* $\mathsf{StackIndex}_{\bar{\alpha}}$ *and* $\mathsf{StackIndex}_{\alpha}$ *with $\bar{\alpha} = 0.3$ and $\alpha = 0.2$. Fig. 2(c) and Fig. 2 (d) demonstrates the detailed process of adapting* StackIndex *from $\bar{\alpha} = 0.3$ to $\alpha = 0.2$.*

First, the algorithm samples a new root set $\rho'_{\alpha} = \{v_5\}$ from the original root set $\rho_{\bar{\alpha}} = \{v_5, v_6\}$. At this stage, all nodes connected to v_6 are classified as Type (1) nodes, v_5 becomes a Type (2) node, and the remaining nodes are classified as Type (3) nodes. Next, the outgoing edge of $v_5 \in \rho'_{\alpha}$ is resampled to v_4, forming a cycle (v_4, v_5). Consequently, both v_4 and v_5 are converted to Type (2) nodes. The algorithm then continues the loop-erased α-random walk from v_4 and v_5 until it reaches v_3, which connects to v_r with a probability of $\alpha = 0.2$. At this point, all nodes connected to v_3 are converted to Type (1) nodes. This process continues until all nodes are classified as Type (1) nodes.

Algorithm 3: The Proposed α-adapting Algorithm

```
Input: Graph $G$, meta-index StackIndex$_{\bar{\alpha}}$, new $\alpha \le \bar{\alpha}$
Output: New index StackIndex$_{\alpha}$
1  $p \leftarrow 1 - \frac{\alpha}{\bar{\alpha}}$;
2  for $i = 1, \cdots, \omega$ do
3      StackForest = (Next, Root) $\leftarrow$ StackIndex$_{\bar{\alpha}}[i]$;
4      $Q \leftarrow$ Sample $p$ proportion from $\rho$(StackForest$_{\bar{\alpha}}$);
5      Update each node's Type accordingly;
       // Continue loop-erased $\alpha$-random walk
6      while $Q$ not empty do
7          $u \leftarrow Q.pop()$;
8          Resample $u$'s neighbor, record to Next$[u]$;
9          if Next$[u] = v_r$ then
10             Set $u$'s Type to 3, set Root to $u$;
11             break;
12         $c \leftarrow u$;
13         while $c$ walk along Next do
14             if $c = u$ then
15                 Push past nodes to $Q$, set Type to 1;
16             else if Type$[c] = 3$ then
17                 Set past nodes' Type to 3, set Root to Root$[c]$;
18             if $c = u$ or Type$[c] \ne 2$ then break;
       // Update Root
19     for $u \in V$ s.t. Type$[u] = 2$ do
20         $c \leftarrow u$;
21         while $c$ walk along Next do
22             if Type$[c] = 3$ then
23                 Set past nodes' Type to 3, set Root to Root$[c]$;
24                 break;
25     StackIndex$_{\alpha}[i] \leftarrow$ StackForest;
26 return New index StackIndex$_{\alpha}$;
```

During the adaptation process, the newly resampled nodes in the trajectory are highlighted in yellow, and two new cycles are popped. The Next *vector is maintained as the top of the* Stack *throughout the process. After the loop-erased α-random walk completes, the* Root *vector is updated to ensure correctness.*

Theoretical analysis of Algorithm 3. We give theoretical analysis on the adapting-α algorithm. We first prove the important property that the resulting spanning forest is independent and the order in which nodes are resampled first (Lemma 4.3 and Lemma 4.4). This helps us to build the correctness of Algorithm 3 (Theorem 4.5). Then, we also derive the time complexity of Algorithm 3 (Theorem 4.6). Throughout this analysis, we use notations assuming that G is weighted. In the case where G is unweighted, the weight of each edge is set to 1. We denote the weight of an edge (v_s, v_t) by w_{st} or $w(v_s, v_t)$, and the weight of a set of edges S is denoted as $w(S) = \prod_{(v_i,v_j)\in S} w(v_i, v_j)$.

Lemma 4.3. *(Order Irrelevant Property [43]) If Algorithm 2 successfully terminates, the order in which out-neighbors are sampled does not affect the final* StackIndex.

Another crucial property of StackIndex is that the resulting spanning forest stored in Next and Root is independent of intermediate popped cycles (i.e., elements of Stack except for the top), even if we condition on a particular set of cycles being popped (Lemma 4.4). Combining with Lemma 4.3, we can know that there is no bias to the final spanning forest even if we start with certain loops already discovered.

Algorithm 4: Query Processing with StackIndex

```
Input: Graph $G$, source node $s$, damping factor $\alpha$, meta-index StackIndex$_{\bar{\alpha}}$, error parameter $\epsilon$
Output: $\hat{\pi}(s,u)$ as the approximation of $\pi(s,u)$ for $u \in V$
1  Initialize $\hat{\pi}(s,u) = 0, r_s(u) = 1$ for $u \in V$;
2  StackIndex$_\alpha$ ← $\alpha$-adapting($G$, StackIndex$_{\bar{\alpha}}$, $\alpha$);
3  $r_{max} \leftarrow \sqrt{\frac{1}{\alpha\omega}}$;
4  if $r_{max} < 1$ then
5      $\hat{\pi}(s,\cdot), r_s(\cdot) \leftarrow$ Forward-Push$(G, s, \alpha, r_{max})$;
6      for $i = 1 : \lceil r_{max}\omega \rceil$ do
7          StackForest = (Next, Root) ← StackIndex$_\alpha[i]$;
8          for $u \in V$ do
9              $\hat{\pi}(s,u) += r_s[\mathrm{Root}[u]]/\lceil r_{max}\omega \rceil$;
10 else
11     for $i = 1 : \omega$ do
12         StackForest = (Next, Root) ← StackIndex$_\alpha[i]$;
13         for $u \in V$ do
14             $\hat{\pi}(s,u) += r_s[\mathrm{Root}[u]]/\omega$;
15 return $\hat{\pi}(s,u)$ for $u \in V$;
```

LEMMA 4.4. *(Independence Property) Let Φ be the spanning forest corresponding to the* StackForest, *and $\mathscr{C}$ be the set of cycles popped during the process. Then, we have:*

$$Pr(\Phi = \phi, \mathscr{C} = C) \propto w(\phi) \cdot w(C),$$
$$Pr(\Phi = \phi, \mathscr{C} = C \mid \tilde{C} \subseteq \mathscr{C}) \propto w(\phi) \cdot w(C \backslash \tilde{C}).$$

According to Lemma 4.3 and Lemma 4.4, we can derive the correctness of Algorithm 3 as follows.

THEOREM 4.5. *(Correctness of Algorithm 3) Each* StackForest *in* StackIndex$_\alpha$ *returned by Algorithm 3 is a uniform α-spanning forest.*

Essentially, we keep each spanning forest uniform in new α setting by resampling root set and continuing loop-erased α-random walk. The root distribution of the updated α-spanning forest remains an unbiased estimator of PPR. In addition, as the sample size needed to guarantee an ϵ-error guarantee for smaller α becomes larger (due to the parameter setting of the Forward Push stage), in our query algorithm, we read more samples to ensure a certain ϵ-relative error is guaranteed. No matter how α changes, there is an upper bound on the sample size, which is exactly the sample size of our meta-index. Thus, the process does not amplify errors, even for very small or large values of α (See also Lemma 4.7).

Below, we analyze the time complexity of Algorithm 3.

THEOREM 4.6. *(Time Complexity of α-adapting) The time complexity of Algorithm 3 is $O(\omega \cdot (\mathrm{Tr}(I - \frac{P}{\beta+1})^{-1} - \mathrm{Tr}(I - \frac{P}{\bar{\beta}+1})^{-1} + n))$, where $\bar{\beta} = \frac{\bar{\alpha}}{1-\bar{\alpha}}$.*

Note that the first term $\mathrm{Tr}(I - \frac{P}{\beta+1})^{-1} - \mathrm{Tr}(I - \frac{P}{\bar{\beta}+1})^{-1}$, is typically much smaller than $\mathrm{Tr}(I - \frac{P}{\beta+1})^{-1}$ in real-life graphs, as shown in Appendix A.2, table 6. This observation suggests that the proposed α-adapting method is significantly faster than the approach based on rebuilding the index with α.

4.3 The Query Processing Algorithm

After adapting meta index StackIndex$_{\bar{\alpha}}$ to StackIndex$_\alpha$, the querying process continues by invoking Forward Push. We can use the approach in [44] to enhance theefficiency of Forward Push.

Algorithm 4 outlines the pseudo-code for the whole query processing algorithm. To approximate the PPR vector $\pi(s,\cdot)$ for a source node s and a damping factor $\alpha \leq \bar{\alpha}$, the algorithm begins with a

Table 2. Complexity comparison of different PPR Algorithms, $\tau_\alpha = \mathrm{Tr}\left((I - \frac{P}{\beta+1})^{-1}\right)$, $\beta = \frac{\alpha}{1-\alpha}$ is the expected number of steps in a loop-erased α-random walk, $\tau_\alpha \leq \frac{n}{\alpha}$.

Method	**Category**	**Index Building Time**	**Query Time**	**Query New α Time**	**Space**
SpeedPPR [44]	Index-free	-	$O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$	$O\left(\frac{n\log n}{\alpha\epsilon}\right)$	-
SpeedL [26]	Index-free	-	$O\left(\frac{\sqrt{n}\log n}{\epsilon}\sqrt{\frac{\tau_{\bar{\alpha}}}{\bar{\alpha}}}\right)$	$O\left(\frac{\sqrt{n}\log n}{\epsilon}\sqrt{\frac{\tau_\alpha}{\alpha}}\right)$	-
SpeedPPR+ [44]	Index	$O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$	$O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$	$O\left(\frac{n\log n}{\alpha\epsilon}\right)$	$O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$
SpeedL+ [26]	Index	$O\left(\frac{\sqrt{n}\log n}{\epsilon}\sqrt{\frac{\tau_{\bar{\alpha}}}{\bar{\alpha}}}\right) \leq O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$	$O\left(\frac{\sqrt{n}\log n}{\epsilon}\sqrt{\frac{\tau_{\bar{\alpha}}}{\bar{\alpha}}}\right)$	$O\left(\frac{\sqrt{n}\log n}{\epsilon}\sqrt{\frac{\tau_\alpha}{\alpha}}\right)$	$O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$
StackIndex	Index	$O\left(\frac{\sqrt{n}\log n}{\epsilon}\sqrt{\frac{\tau_{\bar{\alpha}}}{\bar{\alpha}}}\right) \leq O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$	$O\left(\frac{\sqrt{n}\log n}{\epsilon}\sqrt{\frac{\tau_{\bar{\alpha}}}{\bar{\alpha}}}\right)$	$O\left(\frac{\sqrt{n}\log n}{\epsilon}\sqrt{\frac{(\tau_\alpha-\tau_{\bar{\alpha}})}{\alpha}}\right)$	$O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$

forward-push (Algorithm 1) on the graph G from the source node s, generating the vectors q_s and r_s (lines 1-2). The initial approximation $\hat{\pi}(s,\cdot)$ is set to q_s. In the refinement phase, the algorithm updates each $\hat{\pi}(s,u)$ by adding the scaled sum of residues for nodes in the spanning tree component containing u. The sum is weighted by the degree of u and normalized by the total degree of nodes in T (lines 3-7). After processing all elements, the algorithm returns the approximated PPR vector $\hat{\pi}(s,\cdot)$ (line 8).

Lemma 4.7. *Algorithm 4 returns $\hat{\pi}(s,u)$ such that $|\hat{\pi}(s,u) - \pi(s,u)| \leq \min(1, \sqrt{\frac{1}{\alpha W}})\epsilon\pi(s,u)$ for all $u \in V$ and $\epsilon > 0$ with probability at least $1-\delta$.*

From Lemma 4.7, we can see that when α becomes smaller, the relative error bound $\sqrt{\frac{1}{\alpha W}}\epsilon$ can grow, but our parameter setting ensures that the relative error cannot exceed ϵ. This results in the correctness of Algorithm 4.

Theorem 4.8. *(Correctness of Algorithm 4) $\hat{\pi}(s,\cdot)$ returned by Algorithm 4 satisfies relative-error guarantee.*

Theorem 4.9. *(Time Complexity of Algorithm 4) The time complexity for querying under adapted* StackIndex$_\alpha$ *is $O(\min(\frac{\sqrt{\omega n}}{\sqrt{\alpha}}, \omega n))$. Thus the overall time complexity is $O(\omega \cdot (\mathrm{Tr}(I - \frac{P}{\beta+1})^{-1} - \mathrm{Tr}(I - \frac{P}{\bar{\beta}+1})^{-1} + n)))$*

Complexity comparison. To fairly compare the time and space complexity of different algorithms under various scenarios, following prior studies [26, 41, 44], we assume the graph is scale-free, i.e., $m = O(n\log n)$ and $\Delta_{out} = O(n\log n)$. Additionally, we assume $\delta = \frac{1}{n}$ and $p_f = \frac{1}{n}$. Similar to [26, 41, 44], we minimize query complexity by balancing the time of sampling and push. The complexities of SpeedPPR and SpeedL+ have been discussed in [44]. For SpeedL, SpeedL+, and StackIndex, the push time is $O\left(\frac{\log n}{\alpha r_{max}}\right)$, and the sampling time is $O(r_{max}W\tau_\alpha)$, where r_{max} is the push parameter, τ_α stands for the sampling time of a uniform rooted α-spanning forest which is less than or equal to $\frac{n}{\alpha}$. After balancing, the query time complexity for these methods is $O\left(\sqrt{\log nW} \cdot \sqrt{\frac{\tau_\alpha}{\alpha}}\right) = O\left(\frac{\sqrt{n}\log n}{\epsilon} \cdot \sqrt{\frac{\tau_\alpha}{\alpha}}\right)$. For StackIndex, when querying a new α, the once-sampling time changes from τ_α to $\tau_\alpha - \tau_{\bar{\alpha}}$, resulting in a query time complexity of $O\left(\frac{\sqrt{n}\log n}{\epsilon} \cdot \sqrt{\frac{\tau_\alpha - \tau_{\bar{\alpha}}}{\alpha}}\right)$. These results are presented in Table 2, which illustrates a detailed comparison of the space and time complexity of StackIndex with existing methods. In summary: (i) Compared to index-free algorithms, StackIndex requires an additional $O\left(\frac{n\log n}{\bar{\alpha}\epsilon}\right)$ space, while the query time complexity remains the same as index-free methods like SpeedL. However, since random walk and random spanning forest samples are pre-stored as an index, the empirical query performance of index-based solutions is significantly better than index-free solutions. This observation aligns with prior findings in [26, 41, 44]. (ii) Compared to other index-based algorithms, StackIndex has the same space complexity (differing only in constant factors, e.g., $2n$ vs. n). However, the query time complexity for a new α in StackIndex is significantly lower, making it highly efficient for handling PPR queries with varying α values.

5 Maintaining the StackIndex

In this section, we address the problem of maintaining the StackIndex in dynamic graphs. Note that vertex addition or deletion can be treated as a series of edge insertion or deletion operations, we focus mainly on edge insertions and deletions. To handle edge updates, we need to additionally store the stack structure. Each stack is represented as Stack, which captures the trajectory of a loop-erased walk (line 9 of Algorithm 2). Note that including Stack in our meta-index StackIndex does not introduce significant space overhead, as demonstrated in Appendix A.2. This ensures that the additional storage required for maintaining the stack structures remains manageable, preserving the overall efficiency of the StackIndex approach. An example of the extended StackIndex structure can be found in Fig. 1(c), where the stacks are illustrated. For example, for node v_1, in addition to Next$[v_1]$ and Root$[v_1]$, it also stores Stack$[v_1] = \{v_2, v_4, v_2, v_4\}$.

Since we estimate PPR according to random α-spanning forests sampled by loop-erased α-random walk, maintaining StackIndex can be translated to maintaining the loop-erased walk trajectory recorded in the stacks. However, updating a loop-erased walk presents a significant challenge, as modifying one loop-erased walk trajectory may affect others. A naive approach is to rebuild the entire index for each updated edge, which is clearly inefficient. Below, we propose a novel approach that can significantly reduce redundant rebuilding operations by including Stack into the index.

The high-level idea of our method is, when updating an edge (u, v), only the loop-erased α-random walk trajectory that passes this edge is affected, while the other trajectory that does not pass u can be preserved. Notice that Stack$_u$ records the edges that u passes through during the loop-erased walk. If the update edge is not included in Stack$_u$, the update can be safely skipped. Otherwise, we simply resample the stack structure. We design update algorithms for both edge insertion and edge deletion. Theoretical analysis and experiments both show that the improvement is substantial after resampling is skipped.

Edge insertion. Assume that an edge (u, v) is inserted, the pseudo-code of the Edge-Insertion algorithm is shown in Algorithm 5. Specifically, it starts by sampling a “first occurrence” variable, which follows a geometric distribution with parameter $\frac{1}{d_u+1}$ (line 3). This variable represents the first time the edge (u, v) is sampled in Stack$_u$ after (u, v) is inserted. If the “first occurrence” exceeds the length of Stack$_u$, there is no need to update, as none of the edges stored in Stack$_u$ need to be modified (lines 4-5). Otherwise, we update the edge at “first occurrence” position to v, and change the edge following “first occurrence” position to v with probability $\frac{1}{d_u+1}$ (lines 6-8). Finally, we rebuild the StackForest from updated Stack (line 9). Our theoretical analysis shows that this will produce an unbiased random spanning forest after edge insertion, while significantly reducing the update time compared to resampling from scratch.

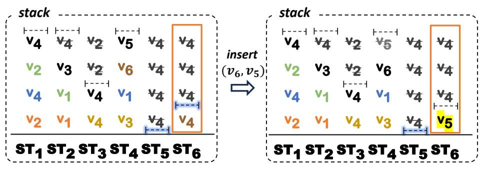
Edge deletion. When an edge (u, v) is deleted, similar to Edge-Insertion, the Edge-Deletion algorithm is outlined in Algorithm 6. For each node $p \in$ Stack$_u$, the algorithm uniformly replaces v to another out-neighbor with probability $\frac{1}{d_u-1}$ (lines 3-5). If v is not found in Stack$_u$, there is no need to update StackForest (lines 6-7). Otherwise, we rebuild StackForest to update Stack (line 8). Our theoretical analysis shows that it can produce an unbiased spanning forest after the edge is deleted.

Algorithm 5: Edge-Insertion

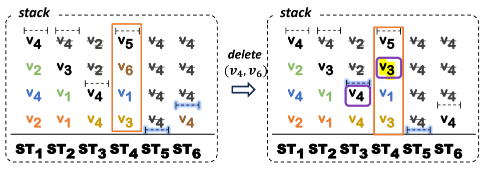
```
   Input: Graph $G$, index StackIndex($G$), insert edge $e = (u, v)$
   Output: New index StackIndex($G \cup e$)
1  for $i = 1, 2, \cdots, \omega$ do
2      StackForest = (Next, Root, Stack) ← StackIndex($G$)[$i$];
3      Sample $firstPos \sim$ Geom($\frac{1}{d_u+1}$);
4      if $firstPos >$ Stack$_u$.length() then
5          continue;
6      Replace Stack$_u$[$firstPos$] to $v$;
7      foreach $p \in$ Stack$_u$[$firstPos + 1$ :] do
8          Replace $p$ with $v$ with probability $\frac{1}{d_u+1}$;
9      Rebuild Next and Root;
10     StackIndex($G \cup e$)[$i$] ← StackForest;
11 return StackIndex($G \cup e$)
```

Algorithm 6: Edge-Deletion

```
   Input: Graph $G$, index StackIndex($G$), delete edge $e = (u, v)$
   Output: New index StackIndex($G \backslash e$)
1  for $i = 1, 2, \cdots, \omega$ do
2      StackForest = (Next, Root, Stack) ← StackIndex($G$)[$i$];
3      foreach $p \in$ Stack$_u$ do
4          if $p = v$ then
5              Replace $p$ with other out-neighbors of $u$ with probability $\frac{1}{d_u-1}$;
6      if no replacement is done then
7          continue;
8      Rebuild Next and Root;
9      StackIndex($G \backslash e$)[$i$] ← StackForest;
10 return StackIndex($G \backslash e$)
```



(a) Edge insertion



(b) Edge deletion

Fig. 3. An illustrative example of inserting edge (v_6, v_5) and deleting edge (v_4, v_6).

Example 4. *Fig. 3 illustrates an example of how we maintain the* StackIndex. *The example graph is shown in Fig. 1(a). When inserting edge (v_6, v_5), we first sample a random variable "firstPos" from geometric distribution with parameter $\frac{1}{d_6+1} = 0.5$ and get 1 (If "firstPos" is larger than the length of* Stack$_6$*, which is 1 in our example, we stop updating). Therefore, we replace* Stack$_6$[1] *to v_5. Then, we run loop-erased α-random walks that use out-neighbors pre-stored in* Stack *if available from each node. During this process,* Stack *is updated if there exist new resampled records and* Next *and* Root *are rebuilt.*

When deleting edge (v_4, v_6), we first examine whether v_6 exists in Stack$_4$ *(if we are to delete (v_4, v_2), since v_2 does not appear in* Stack$_4$*, we stop updating). We sample an out-neighbor of v_4 and replace v_6 to v_5. Then, we run loop-erased α-random walks that use out-neighbors pre-stored in* Stack *if available from each node. During this process,* Stack *is updated if there are new resampled records and* Next *and* Root *are rebuilt.*

Theoretical analysis of our algorithms. We analyze the correctness and complexity of Algorithm 5 and Algorithm 6 as follows.

Theorem 5.1. *(Correctness of Algorithm 5 and Algorithm 6) For edge insertion (deletion),* $\mathsf{StackIndex}(G\cup e)$ *(*$\mathsf{StackIndex}(G\backslash e)$*) provided by Algorithm 5 (Algorithm 6) is unbiased if* $\mathsf{StackIndex}(G)$ *is unbiased.*

To analyze the improvement over the naive method, the key is to bound the probability of satisfying our *skipping condition.* By skipping condition, we mean that the "first occurrence" exceeds Stack_u's length in edge-insertion, or that v is not found in Stack_u in edge-deletion.

For the edge-insertion case, let X_u denote the stack length for node u, and Y_u denote the "first occurrence" position. The skipping condition is satisfied when $X_u < Y_u$. For the edge-deletion case, let Z_{uv} denote the first position where v appears in Stack_u. The skipping condition is satisfied when $X_u < Z_{uv}$. Since each element in Stack_u is uniformly sampled, $Z_{uv} - 1$ follows a geometric distribution with parameter $\frac{1}{d_u}$. Note that Y_u follows a geometric distribution with parameter $\frac{1}{d_u+1}$, and Y_u and Z_{uv} have the same expectation. In the following, we focus mainly on analyzing the edge-insertion case, and the same result can be easily derived for the edge-deletion case.

Since Y_u is positive and X_u is non-negative, and X_u and Y_u are independent, we can use the classic Markov's inequality to derive:

$$P\left(\frac{X_u}{Y_u} < 1\right) = 1 - P\left(\frac{X_u}{Y_u} \geq 1\right) \geq 1 - \mathbb{E}\left(\frac{X_u}{Y_u}\right) = 1 - \frac{\mathbb{E}(X_u)}{\mathbb{E}(Y_u)} = 1 - \frac{\frac{1}{\alpha}\pi(u,u)}{d_u + 1}.$$

The last equality follows from the loop-erased α-random walk explanation of $\pi(u,u)$ (see Section 3). Since $\pi(u,u)$ is a global quantity while d_u is local, it is challenging to derive a clear enhancement bound for each u. However, if u is sampled by degree (referred to as *random arrival model* in [18]), the global expected enhancement can be lower-bounded by:

$$\sum_{u\in V} \frac{d_u}{m} P\left(\frac{X_u}{Y_u} < 1\right) \geq 1 - \sum_{u\in V} \frac{\frac{1}{\alpha}\pi(u,u)}{m} \geq 1 - \frac{n}{\alpha m} = 1 - \frac{1}{\alpha \bar{d}}.$$

This result suggests that datasets with larger average degrees tend to have higher lower bounds for the expected enhancement. However, it is important to note that this is merely a lower bound for the anticipated improvement, and in practice, the actual performance is often significantly better than this bound (Table 3 in Section 6.1). Specifically, our experimental results show that this method can substantially accelerate the update speed, achieving an improvement of over 80% in real-world networks (see Section 6.2).

Theorem 5.2. *The expected running time of Algorithm 5 and 6 under random arrival model [18] is* $O(\omega \cdot \frac{\left(\mathsf{Tr}\left(I - \frac{P}{1+\beta}\right)^{-1}\right)^2}{m})$*.*

In real-life graphs, since $\mathsf{Tr}\left(I - \frac{P}{1+\beta}\right)^{-1}$ is roughly $O(n)$, the time complexity of Algorithm 5 and Algorithm 6 is around $O(\omega \cdot \frac{n^2}{m}) = O(\omega \frac{n}{\bar{d}})$, where $\bar{d} = \frac{m}{n}$ is the average out degree of the graph G.

6 Experimental Evaluation

6.1 Experimental Setup

Algorithms and implementations. To evaluate the performance of the proposed solutions, we implement our index-based methods, including the index building algorithm in Algorithm 2 and query processing in Algorithm 4, denoted by StackIndex. For the baseline methods, we mainly

Table 3. Summary of the datasets used in our experiments

Dataset	n	m	$\bar{d}$	$\mathrm{Tr}(I - \frac{P}{1+\beta})^{-1}$	$1 - \sum_{u \in V} \frac{\frac{1}{\alpha}\pi(u,u)}{m}$
DBLP [23]	317,080	2,099,732	6.62	624,389	0.70
Youtube [23]	1,134,890	5,975,248	5.27	2,202,832	0.63
Pokec [23]	1,632,803	44,603,928	27.32	2,989,787	0.93
Orkut [23]	3,072,441	234,370,166	76.28	5,562,469	0.98
LiveJournal [23]	3,997,962	69,362,378	17.35	7,473,188	0.89

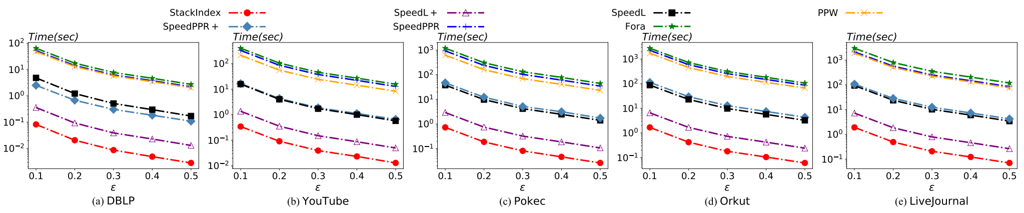


Fig. 4. Query processing time of different algorithms

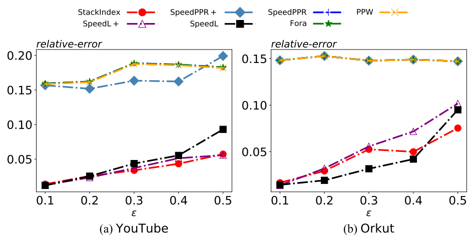


Fig. 5. Accuracy of different algorithms in terms of the maximum relative error metric

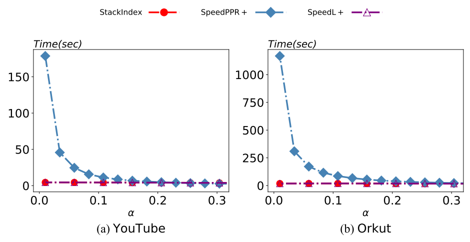


Fig. 6. (Meta-)Index construction time of different algorithms

select two categories of state-of-the-art (SOTA) algorithms for comparison: *online two-stage methods* and *index-based methods*. The former can handle different α settings with low query efficiency, while the latter require rebuilding the indices whenever α varies. For *online methods*, we select four SOTA methods, including the random walk sampling-based approach SpeedPPR [44] and the random spanning forest sampling-based approach SpeedL [26], as well as FORA [41] and its variance-reduced variant PPW [25]. For *index-based methods*, we select the index-based methods SpeedPPR+ and SpeedL+ proposed in [44] and [26], respectively. Note that another index-based solution exists [40], but it is less efficient than SpeedPPR+ [44] and SpeedL+ [26], thus we do not include it in our experiments. For all these methods, there is only one parameter ϵ, which we set by default as 0.5 following [26, 44]. To handle edge updates, we implement Algorithm 5 and 6 for StackIndex. We compare it with the SOTA index maintenance method FIRM [18] which is built on the random walk sampling-based method SpeedPPR. All experiments are conducted on an Ubuntu 22.04 server equipped with 128 GB of memory and an AMD 3990X CPU. The proposed algorithms are implemented in C++ and compiled using GCC 9.3.0 with the -O3 optimization. The original C++ implementations of these baselines are used [25, 26, 41, 44].

Accuracy metrics. Note that all algorithms in our experiments can achieve ϵ-relative error guarantees in theory. To measure the accuracy of different algorithms, we adopt the $L1$-error metric, defined as $\sum_{u \in V} |\hat{\pi}(s,u) - \pi(s,u)|$, which is widely used in previous studies [26, 41, 44]. Additionally, we also use the max relative error as an alternative metric, defined as $\max_{u \in V} |\hat{\pi}(s,u) - \pi(s,u)|$.

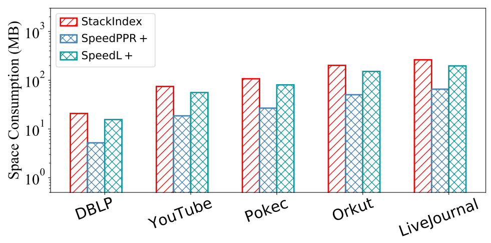


Fig. 7. Index space of different algorithms ($\alpha = 0.2$).

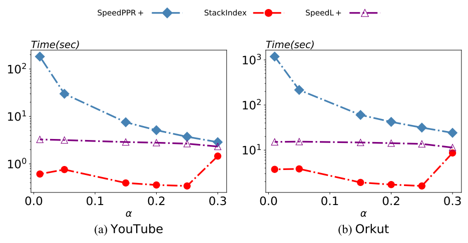


Fig. 8. Comparison of the indexing (transforming) time of different algorithms with varying α

Datasets and query generations. We conduct extensive experiments on five real-world datasets (Table 3): DBLP, Youtube, Pokec, Orkut, and LiveJournal, which are widely used benchmarks in prior studies on PPR computation [26, 27, 38, 41, 44]. Each query is represented as a pair (s, α), where s is the source node and $\alpha \leq \bar{\alpha}$ is the damping factor. In our experiments, we select six α values sampled from a Poisson distribution with a mean of 0.1, covering both small and large damping factors commonly encountered in real-world applications [26, 35, 41, 44]. To ensure robustness, for each α, we generate 1,000 queries by randomly selecting source nodes s from V. This results in a total of 6,000 queries (i.e., 6,000 (s, α) pairs) across all α values. The final results are reported as the average over all queries. Unless otherwise stated, we set $\alpha = 0.2$ (for fixed α experiments) and $\epsilon = 0.5$ as default parameters, following [26, 41, 44], and construct a meta-index with $\bar{\alpha} = 0.3$. For ground-truth results, we compute exact personalized PageRank values using the classic power method with an $L1$ error threshold of 10^{-9} [26, 44].

6.2 Experimental Results

Exp 1: Overall query performance. The results of query processing time of different algorithms are shown in Fig. 4. It is worth emphasizing that the query processing time reported in Fig. 4 for our StackIndex method already incorporates the time taken to transform the meta-index into the desired index using the query parameter α. As can be seen, StackIndex achieves the lowest query time across all five datasets, followed by SpeedL+. SpeedPPR+ performs comparably to SpeedL, while SpeedPPR has the highest query time. The reason is that StackIndex does not require rebuilding for smaller α values after constructing the meta-index, thus it is expected to outperform other indexed methods. FORA and SpeedPPR perform worst due to it is inefficient when handling small α, PPW performs slightly better than SpeedPPR but worse than index-based methods. In general, StackIndex demonstrates approximately a 3× improvement over SpeedL+ and at least a 10× improvement over SpeedPPR+. For example, in the Pokec dataset with $\epsilon = 0.3$, StackIndex requires only 0.08 seconds, while SpeedL+ takes 0.32 seconds and SpeedPPR+ takes 5.13 seconds. These results demonstrate the high efficiency of the proposed solution.

The result of the accuracy of various algorithms are reported in Fig. 5. Due to space limits, the result of $L1$-error is moved to Appendix A.2. As expected, the maximum relative error of StackIndex, SpeedL, and SpeedL+ are very close. This is because all three algorithms are based on the random spanning forest sampling-based method. Additionally, StackIndex, SpeedL, and SpeedL+ exhibit significantly higher accuracy compared to FORA, PPW, SpeedPPR and SpeedPPR+. This is because random spanning forest-based solutions are generally more accurate than random walk sampling-based solutions for small α values, and remain comparable for larger α. These findings are consistent with previous results observed in [26]. In conclusion, these results provide evidence

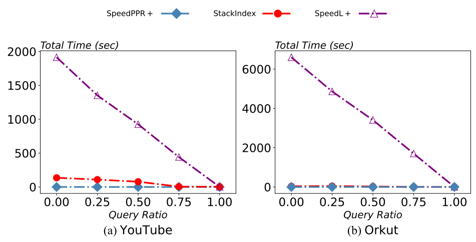


Fig. 9. Index updating performance of different algorithms on the dynamic graph under different workloads

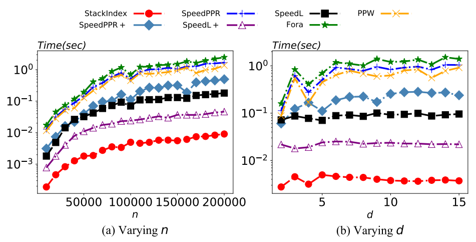


Fig. 10. Query time on synthetic datasets varying n and d

that our StackIndex outperforms existing *index* and *online* methods in terms of both query time and accuracy.

Exp 2: Index construction time and space. We compared the index construction time and space requirements of StackIndex with those of SpeedPPR+ and SpeedL+. It is worth noting that this experiment involves building the meta-index for StackIndex, while for other index-based methods, it involves constructing the index for a specific α. However, StackIndex only needs to build the meta-index once, as it can directly adapt to new α values during queries. In contrast, other methods require reconstructing the index each time a new α is encountered. The results are shown in Figs. 6 and 7, respectively. As shown in Fig. 6, the index construction time for StackIndex and SpeedL+ is nearly identical, as both algorithms employ the Wilson algorithm for sampling random spanning forests. In general, the index construction time for both StackIndex and SpeedL+ is robust with respect to α, given that the Wilson algorithm is insensitive to α, a phenomenon also observed in [26]. Notably, when α is small, the index construction time of StackIndex and SpeedL+ is significantly lower than that of SpeedPPR+, and comparable when $\alpha \geq 0.2$.

Regarding index space (Fig. 7), StackIndex requires slightly more space than SpeedPPR+ and SpeedL+ due to the additional auxiliary information it stores. However, this remains manageable in practice, because the space complexity of StackIndex is $O(\omega n)$. For instance, in the Orkut dataset, StackIndex requires only 201 MB while storing the Orkut graph itself requires 895 MB. These results show that our StackIndex is highly efficient on time and space.

Exp 3: Runtime and accuracy of indexing algorithms with varying α. Recall that our StackIndex method requires transforming the meta-index into the desired index for a given α when processing a query. Specifically, we first construct a meta-index using a damping factor of $\bar{\alpha} = 0.3$ (as most applications typically use α values below 0.3). For a query (s, α), we transform the meta-index into the desired index with the damping factor α and then use this transformed index to answer the query. In this experiment, we evaluate the runtime of this index transformation procedure. For baseline index-based methods such as SpeedPPR+ and SpeedL+, changing the parameter α requires rebuilding the index for the new α. In contrast, StackIndex only needs to transform the existing meta-index to the new α. Fig. 8 shows the runtime of different indexing algorithms with varying α values. Due to space limits, the result of $L1$-error and maximum relative error is moved to Appendix A.2. For StackIndex, the runtime at $\alpha = 0.3$ corresponds to the time required to construct the meta-index, while the runtime for $\alpha < 0.3$ corresponds to the time needed to transform the meta-index to the new α. As expected, StackIndex demonstrates significant performance improvements over SpeedPPR+ and SpeedL+ across all datasets. This is because StackIndex avoids the need to rebuild the index, a costly operation required by both SpeedPPR+ and SpeedL+. By eliminating this overhead, StackIndex achieves greater efficiency in query processing, as confirmed in Fig. 4.

In addition to evaluating the runtime of the index transformation procedure, we also assessed the query time required by different algorithms (including index-free methods) when handling various α values, as a supplement to Exp 1. Due to space limits, this experiment is included in Appendix A.2.

Exp 4: Index updating performance. In this experiment, we evaluate the performance of our meta-index maintenance algorithm for dynamic graphs. We compare it against the state-of-the-art PPR index updating algorithm FIRM [18] and include SpeedL+ as a baseline, which rebuilds the index with each edge update. To ensure a fair and consistent comparison with [18], we use the same evaluation approach for all algorithms. Specifically, for each dataset, we randomly partition the graph into two parts: the first part includes 90% of the edges, which forms the initial graph. Then, we generate workloads consisting of 100 updates and queries combined. Updates can involve either: (i) insertion of an edge randomly chosen from the remaining edges, or (ii) deletion of an edge randomly selected from the initial graph. A workload configured with an update percentage of $a\%$ contains $a\%$ updates and $1 - a\%$ PPR queries. The experiment is tested across five different workloads involving edge insertions, edge deletions, and PPR queries, with varying ratios of PPR queries: 0, 0.25, 0.5, 0.75, and 1. The results are shown in Fig. 9. Note that FIRM is the dynamic version of SpeedPPR+; hence, in Fig. 9, we also use SpeedPPR+ to represent FIRM to maintain consistency with our previous figures. The results, shown in Fig. 9, reveal that (1) the proposed StackIndex updating algorithm consistently outperforms SpeedL+. This is not unexpected, as real-world networks often have high-degree nodes, facilitating the selection of nodes that satisfy the skipping condition which greatly accelerates the update. Specifically, on Orkut, where the average node degree ($\bar{\mathbf{d}} = 76.28$) is much higher than in other datasets, StackIndex is more than 100 times faster than SpeedL+; (2) While SpeedPPR+ (i.e., FIRM) has a theoretically $O(1)$ update time and slightly outperforms StackIndex in raw speed, it cannot directly adapt to different values of α, which is one of the key advantages of StackIndex. Furthermore, the update time of StackIndex remains manageable even in very large datasets, such as Orkut and LiveJournal. These results confirm the high efficiency of the proposed index maintenance algorithm.

Exp 5: Performance on synthetic datasets. To further evaluate the performance of StackIndex, we conducted experiments on synthetic datasets generated using the widely adopted Erdős-Rényi (ER) graph model. In the ER model, a parameter p controls the probability of edge generation. We varied n (number of nodes) and $d = n \cdot p$ (average degree) to generate a series of ER graphs and measured the query time of different methods. The results are presented in Fig. 10.

The findings are consistent with those observed on real-world datasets, where StackIndex consistently outperforms other algorithms. As n increases, the query time grows at a similar rate across all methods, which is expected as larger graphs result in higher computational complexity. In contrast, varying d has a relatively smaller impact on query time, highlighting the local nature of PPR computations. Additionally, we conducted further experiments (similar setting to Exp 1, 2, 3, 4) on graphs generated using the Watts-Strogatz (WS) model, a popular method for creating small-world networks, as well as the ER graph. Due to space constraints, the results of these experiments are provided in Appendix A.2.

Exp 6: Performance with very small and large α. To ensure the robustness and comprehensiveness of our conclusions, we set the meta-damping factor to 0.9 (instead of $\bar{\alpha} = 0.3$ in Exp 1) and re-ran Exp 1. For this experiment, we expanded the range of α to $[0.01, 0.9]$. All other settings remained consistent with Exp 1. The results are shown in Figure 11 and 23. Due to space limits, the result of $L1$-error and maximum relative error is moved to Appendix A.2. We observe the following: (i) StackIndex demonstrates outstanding performance across all datasets, with particularly

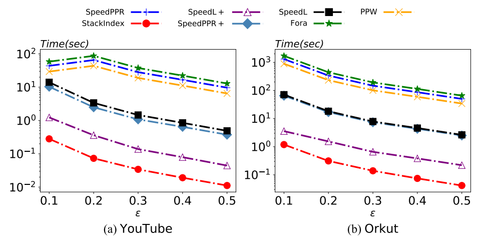


Fig. 11. Query time of different algorithms ($\bar{\alpha} = 0.9$)

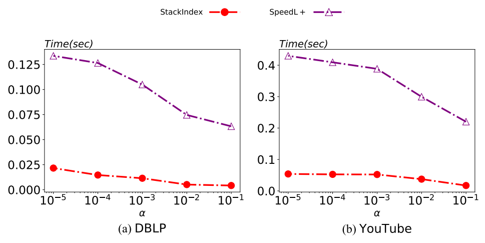


Fig. 12. Query time of different algorithms with very small α

significant improvements for smaller α values; (ii) StackIndex is consistently more than 6× faster than SpeedL+ in most cases and achieves over 100× speedup on the Orkut dataset.

We also conducted experiments with extremely small α values, keeping the meta-damping factor and other parameters at their default settings. The queried α values ranged from $[10^{-1}, 10^{-5}]$. Due to the fact that random walk-based algorithms, such as SpeedPPR, exceeded the 1-hour runtime limit for very small α, we only compared the runtime of StackIndex and SpeedL+, as SpeedL+ demonstrated the closest performance to StackIndex in Exp 1. The results are shown in Figure 12. It can be observed that StackIndex consistently outperforms SpeedL+, achieving at least a 5x speedup. Moreover, for very small α values, StackIndex and SpeedPPR+ significantly outperform random walk-based methods, as also observed in [26]. This demonstrates that StackIndex delivers superior performance even for queries with extremely small α values.

6.3 Case Studies

In this subsection, we present two case studies to demonstrate real-world applications that require the computation of personalized PageRank with different values of α. We focus on two applications on real-life networks: (i) node classification by PPR-based scalable GNN; (ii) PPR-based local graph clustering. The main observation is that the performance of these applications can vary depending on α, and the optimal α is often unknown in advance. In such situations, the proposed StackIndex approach is highly efficient for selecting an optimal α.

Application I: Node classification by PPR-based scalable GNN. Node classification is a classic graph learning task. Given a graph G with only a subset of nodes labeled, the goal of node classification is to infer the labels for all nodes in V. The GNN-based methods is the SOTA methods for this problem [17, 47, 49]. However, most GNNs rely on propagating features along graph edges, which can lead to significant computational overhead in large-scale graphs.

To mitigate the efficiency problem, a popular solution is the PPR-based GNNs [7, 22], which offer an alternative by leveraging similarity matrices to represent graph topology, thereby reducing computational costs. In this approach, the similarity matrix is precomputed, and a network independent of the graph structure is trained. Specifically, let Π be the PPR matrix, PPR-based GNNs first pre-compute the matrix Π, then train a neural network to learn the function f such that $\mathbf{Y} = f(\Pi\mathbf{X})$, where $\mathbf{X}$ is the known labels, $\mathbf{Y}$ is the predicted labels for all nodes. Among these methods, the optimal damping factor α for PPR is often set as hyper-parameter, which is unknown prior to training, and different α values may lead to varying model performance.

In this experiment, we evaluate the performance of PPR-based GNNs across different α values. Specifically, we perform a grid search over the range $[0.01, 0.9]$ and compare the F1-scores achieved for each α. During this process, we utilize StackIndex to precompute the PPR matrices. A meta-index with a damping factor of 0.9 is constructed in advance, and as the grid search progresses, StackIndex efficiently queries the PPR matrices for new α values. In contrast, SpeedPPR requires

Table 4. Results of node classification by PPR-based scalable GNN

Dataset	F1-score	Optimal α	Computational Time (secs)	
			StackIndex	SpeedPPR
Cora	0.834	0.2	0.05	3.74
Pubmed	0.806	0.05	0.08	5.87
Citeseer	0.832	0.7	0.04	2.93
PPI	0.959	0.3	1.21	80.56
Yelp	0.642	0.9	22.10	1380.97

Table 5. Results of PPR-based local graph clustering

Dataset	Conductance	Optimal α	Computational Time (secs)	
			StackIndex	SpeedPPR
DBLP	0.2243	0.01	4.58	271.86
Youtube	0.7476	0.1	8.93	507.45
LiveJournal	0.7911	0.2	52.41	3379.75
Orkut	0.7236	0.3	68.54	\
Amazon	0.4322	0.7	4.51	298.80

recomputing the PPR matrix from scratch for each α. After precomputing the PPR matrices, we use them as input features for GNN training and record the optimal F1-scores. The results, including the optimal α values and precomputation times, are summarized in Table 4. A full table of results is provided in Appendix A.3.

We select datasets widely used in prior studies [7]. As shown in Table 4, the optimal α varies significantly across datasets (e.g., 0.05 for Pubmed and 0.9 for Yelp), emphasizing the importance of selecting the best α for each specific dataset. Moreover, StackIndex demonstrates a significant advantage in efficiency, achieving at least a 60-fold speedup over SpeedPPR in precomputing PPR matrices. Notably, the F1-scores obtained with the optimal α values are substantially higher than those achieved with other α values (as demonstrated in the full table in Appendix A.3, highlighting the effectiveness of our approach in improving model performance.

Application II: PPR-**based local graph clustering.** Local graph clustering is another important task in graph analysis. Given a query node s, it aims to find a small cluster node set near s without traverse the whole graph. PPR-based approach [2, 37, 46] very popular due to its high efficiency. Specifically, it first compute approximate PPR vector w.r.t. the node s. After that, a local cluster is obtained from the PPR vector by a sweep cut procedure [1]. However, similar to PPR-based scalable GNN, the optimal α value for local graph clustering is also hard to decide in prior.

In this experiment, we use several datasets that are widely adopted in previous studies [46], and vary α in range $[0.01, 0.9]$ to see the quality of the clustering results. We use conductance to measure the quality of the results. We randomly select 100 communities with at least 10 nodes for each dataset, and randomly select one node from each community (i.e. 100 sampled nodes for each dataset). During this preocess, we ultilize StackIndex to calculate PPR vectors for each sampled node. A meta-index with a damping factor of 0.9 is constructed in advance, and as the grid search progresses, StackIndex efficiently queries the PPR vectors for new α values. In contrast, SpeedPPR requires recomputing the PPR vectors from scratch for each α. The optimal conductance and its corresponding α, as well as compuation time, is shown in Table 5. A full table including other quality evaluation metrics under different αs is deferred to Appendix A.3.

The results demonstrates that the best α for local graph clustering varies across datasets (e.g., 0.01 for DBLP and 0.7 for Amazon). This highlights the importance of selecting an appropriate α for each specific dataset. Moreover, StackIndex precomputes PPR at least 60 times faster than SpeedPPR, emphasizing its efficiency in scenarios requiring multiple α values. Notably, the conductance achieved with the optimal α values is significantly lower than those with other α values (as demonstrated by the full table in Appendix A.3), underscoring the effectiveness of our approach in improving clustering quality.

7 Related Work

PPR **computation.** There exist a number of efficient methods for PPR computation , including matrix decomposition based approaches [12, 20, 30, 34, 50], and push-based methods [1, 2]. Forward Push [2] was first used for single-source PPR queries and the Backward Push algorithm [1] was proposed for single-target PPR queries. Monte-Carlo methods [3] provided approximation guarantees but are often too slow. To overcome this drawback, two-stage methods that combine push algorithms and random walk sampling have been proposed to improve efficiency while maintaining approximation guarantees. For example, Lofgren et al. [29] combined random walk methods with Backward Push to improve performance for pairwise PPR queries. On top of that, FORA [41], ResAcc [27], and SpeedPPR [44] integrated Forward Push with random walks for efficient single-source PPR queries. To further improve efficiency, Wang et al. [38] directly applied randomness to Backward Push for better accuracy-efficiency trade-offs. Liao et al. [26] proposed a spanning forest sampling method for handling small α. Several variance-reduced techniques were proposed to improve the spanning forest sampling [24]. Yang et al. [42] proposed theoretical results for single-source PPR computation. In addition, PageRank is a node centrality closely related to PPR. Recently, Wang et al. [39] established the optimal algorithm for single-node PageRank centrality computation. A recent survey on PPR computation can be found in [45]. However, all these existing PPR methods are designed for a constant α. When the graph structure varies, several studies focus on designing efficient algorithms to handle edge updates. Ohsaka et al. [31] and Zhang et al. [48] developed methods to update the Forward Push results in dynamic graphs. FIRM [18] is an index maintenance method that handles edge updates by maintaining random walk samples. However, none of the dynamic methods are designed for different α. When α is varying, all these methods require computing from scratch.

Damping factor α of PPR. The damping factor α plays a critical role in determining the balance between the source node and the surrounding nodes during the random walk process. Studies have explored its impact on query accuracy and computational efficiency [6, 8, 9, 11, 15, 28, 33, 36]. Higher values of α make the random walk more localized, while lower values spread the walk more broadly [28, 36]. Several works focus on determining the optimal value of α through heuristics or theoretical analysis [6, 11, 15, 33]. Some studies treat PPR as a function of the damping factor and compute the coefficients of each term (corresponding to different powers of the damping factor) using a Krylov-iteration based method, which enables the computation of PPR for all damping factors [8, 9]. However, none of these methods are index-based methods, rendering a low efficiency for PPR query. To our knowledge, this is the first work that studies index-based solutions for handling all α values.

8 Conclusion

In this paper, we propose a novel index-based approach, termed StackIndex, to approximate single-source Personalized PageRank queries with varying damping factor α. The key advantage of our StackIndex method is that it can handle all α values using a single index, eliminating the need to build a separate index for each α value, as required by previous methods. Thus, our solution can significantly reduce the index building time and space when handling different α values. We present a detailed theoretical analysis of our algorithm. Additionally, we also present a new index updating algorithm to handle the case of dynamic graphs. We conduct extensive experiments on five large real-world datasets, and the results show the high efficiency and effectiveness of the proposed solutions.

Acknowledgments

This work is supported by the NSFC Grants U2241211, 62427808 and U24A20255. Rong-Hua Li and Guoren Wang are the corresponding authors of this paper.

References

[1] Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcraft, Vahab S Mirrokni, and Shang-Hua Teng. 2007. Local computation of pagerank contributions. In *WAW*. 150–165.

[2] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local graph partitioning using pagerank vectors. In *FOCS*. 475–486.

[3] Konstantin Avrachenkov, Nelly Litvak, Danil Nemirovsky, and Natalia Osipova. 2007. Monte Carlo methods in PageRank computation: When one iteration is sufficient. *SIAM* 45, 2 (2007), 890–904.

[4] Konstantin Avrachenkov, Nelly Litvak, and Kim Son Pham. 2007. Distribution of PageRank mass among principle components of the web. In *WAW*. 16–28.

[5] Konstantin Avrachenkov, Nelly Litvak, and Kim Son Pham. 2008. A singular perturbation approach for choosing the PageRank damping factor. *Internet Mathematics* 5, 1-2 (2008), 47–69.

[6] Luca Becchetti and Carlos Castillo. 2006. The Distribution of pageRank Follows a Power-Law Only for Particular Values of the Damping Factor. In *WWW*. 941–942.

[7] Aleksandar Bojchevski, Johannes Gasteiger, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling graph neural networks with approximate pagerank. In *KDD*. 2464–2473.

[8] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2005. PageRank as a Function of the Damping Factor. In *WWW*. 557.

[9] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2007. A Deeper Investigation of PageRank as a Function of the Damping Factor. (2007), 19 pages, 408817 bytes.

[10] Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. 2021. Adaptive Universal Generalized PageRank Graph Neural Network. In *ICLR*.

[11] Hwai-Hui Fu, Dennis K. J. Lin, and Hsien-Tang Tsai. 2006. Damping Factor in Google Page Ranking. 22, 5-6 (2006), 431–444.

[12] Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka. 2012. Efficient Personalized Pagerank with Accuracy Assurance. In *KDD*. 15–23.

[13] David F Gleich, Paul G Constantine, Abraham D Flaxman, and Asela Gunawardana. 2010. Tracking the random surfer: empirically measured teleportation parameters in PageRank. In *WWW*. 381–390.

[14] Zoltán Gyöngyi, Hector Garcia-Molina, and Jan Pedersen. 2004. Combating web spam with trustrank. In *VLDB*. 576–587.

[15] Zheng HaoLin, Hu Jin, and Li WeiKai. 2023. PageRank Algorithm Based on Dynamic Damping Factor. In *2023 International Conference on Cyber-Physical Social Intelligence (ICCSI)*. 381–386.

[16] Taher H. Haveliwala. 2003. Topic-Sensitive PageRank: A Context-Sensitive Ranking Algorithm for Web Search. *TKDE* 15, 4 (2003), 784–796.

[17] Xiaobin Hong, Wenzhong Li, Chaoqun Wang, Mingkai Lin, and Sanglu Lu. 2024. Label attentive distillation for GNN-based graph classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 8499–8507.

[18] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized pagerank on evolving graphs with an incremental index-update scheme. *SIGMOD* 1, 1 (2023), 1–26.

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

[20] Jinhong Jung, Namyong Park, Sael Lee, and U Kang. 2017. BePI: Fast and Memory-Efficient Method for Billion-Scale Random Walk with Restart. In *SIGMOD*. 789–804.

[21] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then Propagate: Graph Neural Networks meet Personalized PageRank. In *ICLR*.

[22] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then Propagate: Graph Neural Networks meet Personalized PageRank. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1gL-2A9Ym

[23] Jure Leskovec and Rok Sosivc. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.

[24] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, Hongyang Chen, Hongchao Qin, and Guoren Wang. 2023. Efficient Personalized PageRank Computation: The Power of Variance-Reduced Monte Carlo Approaches. *Proc. ACM Manag. Data* 1, 2 (2023), 160:1–160:26.

[25] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, Hongyang Chen, Hongchao Qin, and Guoren Wang. 2023. Efficient personalized pagerank computation: The power of variance-reduced monte carlo approaches. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[26] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, and Guoren Wang. 2022. Efficient personalized pagerank computation: A spanning forests sampling based approach. In *SIGMOD*. 2048–2061.

[27] 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.

[28] Tiancheng Liu, Yuchen Qian, Xi Chen, and Xiaobai Sun. 2018. Damping Effect on Pagerank Distribution. In *HPEC*. 1–11.

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

[30] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken ichi Kawarabayashi. 2014. Computing Personalized PageRank Quickly by Exploiting Graph Structures. 7, 12 (2014), 1023–1034.

[31] Naoto Ohsaka, Takanori Maehara, and Ken ichi Kawarabayashi. 2015. Efficient pagerank tracking in evolving networks. In *KDD*. 875–884.

[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. The pagerank citation ranking: Bring order to the web. In *WWW*.

[33] Ali Ali Saber, Aso Kamaran Omer, and Noor Kaylan Hamid. 2022. Google Pagerank Algorithm: Using Efficient Damping Factor. 28, 3 (2022), 1633–1639.

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

[35] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. 2016. Parallel local graph clustering. *Proc. VLDB Endow.* 9, 12 (2016), 1041–1052.

[36] Atul Kumar Srivastava, Rakhi Garg, and P. K. Mishra. 2017. Discussion on Damping Factor Value in PageRank Computation. 9, 9 (2017), 19.

[37] Shayan A Tabrizi, Azadeh Shakery, Masoud Asadpour, Maziar Abbasi, and Mohammad Ali Tavallaie. 2013. Personalized pagerank clustering: A graph clustering algorithm based on random walks. *Physica A* 392, 22 (2013), 5772–5785.

[38] Hanzhi Wang, Zhewei Wei, Junhao Gan, Sibo Wang, and Zengfeng Huang. 2020. Personalized pagerank to a target node, revisited. In *KDD*. 657–667.

[39] Hanzhi Wang, Zhewei Wei, Ji-Rong Wen, and Mingji Yang. 2024. Revisiting Local Computation of PageRank: Simple and Optimal. In *STOC*. 911–922.

[40] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. 2016. HubPPR: Effective Indexing for Approximate Personalized PageRank. *Proc. VLDB Endow.* 10, 3 (2016), 205–216.

[41] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: simple and effective approximate single-source personalized pagerank. In *KDD*. 505–514.

[42] Zhewei Wei, Ji-Rong Wen, and Mingji Yang. 2024. Approximating Single-Source Personalized PageRank with Absolute Error Guarantees. In *ICDT*, Vol. 290. 9:1–9:19.

[43] David Bruce Wilson. 1996. Generating random spanning trees more quickly than the cover time. In *STOC*. 296–303.

[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] Mingji Yang, Hanzhi Wang, Zhewei Wei, Sibo Wang, and Ji-Rong Wen. 2024. Efficient Algorithms for Personalized PageRank Computation: A Survey. *TKDE* 36, 9 (2024), 4582–4602.

[46] Renchi Yang, Xiaokui Xiao, Zhewei Wei, Sourav S Bhowmick, Jun Zhao, and Rong-Hua Li. 2019. Efficient estimation of heat kernel pagerank for local clustering. In *Proceedings of the 2019 International Conference on Management of Data*. 1339–1356.

[47] Yunfeng Yu, Longlong Lin, Qiyu Liu, Zeli Wang, Xi Ou, and Tao Jia. 2024. GSD-GNN: Generalizable and Scalable Algorithms for Decoupled Graph Neural Networks. In *ICMR*. 64–72.

[48] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate personalized pagerank on dynamic graphs. In *KDD*. 1315–1324.

[49] Tianxiang Zhao, Xiang Zhang, and Suhang Wang. 2024. Disambiguated node classification with graph neural networks. In *Proceedings of the ACM Web Conference 2024*. 914–923.

[50] Fanwei Zhu, Yuan Fang, Kevin Chen-Chuan Chang, and Jing Ying. 2013. Incremental and Accuracy-Aware Personalized Pagerank through Scheduled Approximation. (2013).

A Appendix

A.1 Missing proofs

Proof for Lemma 4.1

Proof. In Algorithm 2, each iteration of the for loop (line 3) corresponds to a loop-erased α-random walk. Let the trajectory of the loop-erased α-random walk starting from vertex u be $S_u = (S_u^1, \ldots, S_u^{l_u})$. The spanning forest S produced by the algorithm is composed of $S = (S_{v_1}, S_{v_2}, \ldots, S_{v_n})$, where S_{v_1} is the loop-erased α-random walk starting from vertex v_1, and S_{v_k} is the loop-erased α-random walk starting from vertex v_k, with the boundary set as the union of the previous walks, i.e., $B_k = \bigcup_{j=1}^{k-1} S_{v_j}$. For convenience, we denote the first p vertices of S_{v_k} as $S_{v_k}[: p]$, $P_\alpha := \frac{1}{\beta+1}P$. Below, we prove that S satisfies the definition of a uniform rooted α-spanning forest. Since l_{v_1} is determined by v_r, we have:

$$\begin{aligned}
P(S_{v_1}) &= \alpha \cdot \prod_{i=1}^{l_{v_1}-2} (1-\alpha)\frac{w(S_{v_1}^i, S_{v_1}^{i+1})}{d_{\text{out}}(S_{v_1}^i)} \\
&\quad \cdot \Pr(S_{v_1}^i \text{ returns to itself without reaching } B_1 \cup S_{v_1}[: i-1]) \\
&= \alpha(1-\alpha)^{l_{v_1}-2} \prod_{i=1}^{l_{v_1}-2} \frac{w(S_{v_1}^i, S_{v_1}^{i+1})}{d_{\text{out}}(S_{v_1}^i)} \cdot \frac{\det(I-(P_\alpha)_{-B_1\cup S_{v_1}[:i-1]})}{\det(I-(P_\alpha)_{-B_1\cup S_{v_1}[:i]})} \\
&= \alpha(1-\alpha)^{l_{v_1}-2} \cdot \frac{\det(I-(P_\alpha)_{-B_1\cup S_{v_1}})}{\det(I-(P_\alpha)_{-B_1})} \cdot \prod_{i=1}^{l_{v_1}-2} \frac{w(S_{v_1}^i, S_{v_1}^{i+1})}{d_{\text{out}}(S_{v_1}^i)}.
\end{aligned}$$

Similarly, for S_{v_k}, if l_{v_k} is determined by v_r, we have:

$$\begin{aligned}
P(S_{v_k}|S_{v_1}, \ldots, S_{v_{k-1}}) = \alpha(1-\alpha)^{l_{v_k}-2} \cdot \frac{\det(I-(P_\alpha)_{-B_k\cup S_{v_k}})}{\det(I-(P_\alpha)_{-B_k})} \cdot \\
\prod_{i=1}^{l_{v_k}-2} \frac{w(S_{v_k}^i, S_{v_k}^{i+1})}{d_{\text{out}}(S_{v_k}^i)}.
\end{aligned}$$

Otherwise, we have:

$$\begin{aligned}
P(S_{v_k}|S_{v_1}, \ldots, S_{v_{k-1}}) = (1-\alpha)^{l_{v_k}-1} \cdot \frac{\det(I-(P_\alpha)_{-B_k\cup S_{v_k}})}{\det(I-(P_\alpha)_{-B_k})} \cdot \\
\prod_{i=1}^{l_{v_k}-1} \frac{w(S_{v_k}^i, S_{v_k}^{i+1})}{d_{\text{out}}(S_{v_k}^i)}.
\end{aligned}$$

Note that $B_1 = \emptyset$, $B_{k+1} = B_k \cup S_{v_k}$ for $k = 1, 2, \ldots, n-1$, and $B_{v_n} \cup S_{v_n} = V$, $\sum_k (l_{v_k} - 1) = n$. We can express $P(S)$ as follows:

$$
\begin{aligned}
P(S) &= P(S_{v_1}) \cdot P(S_{v_2}|S_{v_1}) \cdot \cdots \cdot P(S_{v_n}|S_{v_1}, \ldots, S_{v_{n-1}}) \\
&= (1-\alpha)^{\sum_k l_{v_k} - 1} \cdot \left(\prod_{r \in \rho(S)} \frac{\alpha}{1-\alpha} \right) \cdot \frac{\det(I - (P_\alpha)_{-B_{v_n} \cup S_{v_n}})}{\det(I - (P_\alpha)_{-B_1})} \cdot \\
&\quad \frac{w(S)}{\prod_{u \in V \setminus \rho(S)} d_{\text{out}}(u)} \\
&= (1-\alpha)^n \cdot \left(\prod_{r \in \rho(S)} \frac{\alpha d_{\text{out}}(r)}{1-\alpha} \right) \cdot \frac{1}{\det(I - P_\alpha)} \cdot \frac{w(S)}{\prod_{u \in V} d_{\text{out}}(u)} \\
&\propto w(S) \cdot \left(\prod_{r \in \rho(S)} \frac{\alpha d_{\text{out}}(r)}{1-\alpha} \right).
\end{aligned}
$$

This completes the proof. □

Proof for Theorem 4.2

Proof. Let $\overline{G} = (\overline{V}, \overline{E})$, where $\overline{V} = V \cup \{v_r\}$ and $\overline{E} = E \cup \{(v_i, v_r) : w(v_i, v_r) = \beta d_{out}(v_i), i \in [n]\}$, with $\beta = \frac{\alpha}{1-\alpha} \in (0, +\infty)$. Algorithm 2 can then be viewed as performing ω samples using Wilson's algorithm to generate spanning trees rooted at v_r in $\overline{G}$. Hence, the time complexity of the algorithm is $O(\omega \cdot \mathsf{Tr}(I - \overline{P}_{-v_r})^{-1})$. Noting that $\overline{P}_{-v_r} = \frac{1}{\beta+1} P$, we derive a time complexity of $O(\omega \cdot \mathsf{Tr}(I - \frac{P}{\beta+1})^{-1})$. This quantity is typically $O(n)$ in real-life graphs [26], which is also verified in our experiments (see Section 6, Table 3). For space, each sample requires storing two additional n-dimensional vectors, so the total space complexity is bounded by $O(n)$. □

Proof for Lemma 4.4

Proof. Since each element in the stacks is drawn independently, the cycles are mutually independent, and the cycles are also independent of the spanning forest. □

Proof for Theorem 4.5

Proof. After resampling the spanning tree of $\mathsf{StackForest}_{\bar{\alpha}}$, a new tree T' is obtained. To demonstrate the uniformity of the resulting structure, we claim that:

$$\forall u, v \neq r, \quad P(T'(u) = v) = (1-\alpha) \frac{w_{uv}}{d_u}, \quad P(T'(u) = v_r) = 1 - \alpha.$$

This assertion holds because, in our processing procedure, the probability of each edge being part of the new tree depends on both the original probabilities in T and the resampling adjustment. Specifically, we have:

$$P(T'(u) = v) = P(T(u) = v) + P(T(u) = v_r) \cdot \left(1 - \frac{\alpha}{\bar{\alpha}}\right) \frac{w_{uv}}{d_u}.$$

To evaluate this, consider the original probabilities in T. We need to compute $P(T(u) = v)$ and $P(T(u) = v_r)$. First, let us calculate $P(T(u) = v)$, which is equivalent to computing $P((u, v) \in T)$. Since T is a uniform rooted $\bar{\alpha}$-spanning forest, it equals the ratio of the total weight of spanning forests containing (u, v) to the total weight of all spanning forests:

$$P(T(u) = v) = \frac{\sum_{\phi:(u,v) \in \phi} w(\phi)}{\sum_\phi w(\phi)}.$$

By factoring out w_{uv} from the spanning forests containing edge (u, v) and simplifying further, we obtain:

$$\begin{aligned}
P(T(u) = v) &= \frac{w_{uv} \cdot \sum_{\phi:(u,v)\in\phi} \prod_{v_i\in V\setminus\{u\}} w(v_i, \phi(v_i))}{\sum_{\phi:(u,p)\in\phi} \prod_{v_i\in V\setminus\{u\}} w(v_i, \phi(v_i))} \\
&\quad \cdot \frac{1}{\bar{\beta} d_u + \sum_{p\in N_{\text{out}}(u)} w_{up}} \\
&= \frac{w_{uv}}{(1+\bar{\beta})d_u} = (1-\bar{\alpha})\frac{w_{uv}}{d_u}.
\end{aligned}$$

The probability of u connecting to the root r is then derived as:

$$\begin{aligned}
P(T(u) = v_r) &= 1 - \sum_{p\in N_{\text{out}}(u)} P(T(u) = p) \\
&= 1 - \frac{1}{1+\bar{\beta}} = \bar{\alpha}.
\end{aligned}$$

Substituting these results into $P(T'(u) = v)$:

$$\begin{aligned}
P(T'(u) = v) &= (1-\bar{\alpha})\frac{w_{uv}}{d_u} + (\bar{\alpha} - \alpha)\frac{w_{uv}}{d_u} \\
&= (1-\alpha)\frac{w_{uv}}{d_u}. \\
P(T'(u) = v_r) &= 1 - \sum_{p\in N_{\text{out}}(u)} P(T'(u) = p) = 1 - \alpha.
\end{aligned}$$

Hence, we confirm that T' satisfies the desired probability distribution. This implies that T' can be considered as the first layer of a new graph stack.

According to Theorem 4.3, $\mathsf{StackIndex}_\alpha$ is independent of the traversal order. Consequently, the nodes in the queue Q after initialization in Algorithm 3 can be interpreted as the last traversed nodes in Algorithm 2. In this way, $\mathsf{StackIndex}_{\bar{\alpha}}$ represents an intermediate state of Algorithm 2.

Moreover, by Theorem 4.4, the cycles that are already removed in $\mathsf{StackIndex}(\bar{\alpha})$ do not affect the uniformity of the spanning forest generated by the adapting algorithm. Therefore, the StackForest obtained after executing Algorithm 3 remains uniform. □

Proof for Theorem 4.6

Proof. $\mathsf{StackIndex}_{\bar{\alpha}}$ can be regarded as an intermediate process of $\mathsf{StackIndex}_\alpha$. The time complexity of Algorithm 3 is thus $O(\omega \cdot (\mathsf{Tr}(I - \frac{P}{\beta+1})^{-1} - \mathsf{Tr}(I - \frac{P}{\bar{\beta}+1})^{-1}) + n)$, plus the time for initializing the status vector and updating Root, which is $O(n)$. □

Proof for Lemma 4.7

Proof. According to [41], we have $Pr(|\tilde{\pi}(s,u) - \pi(s,u)| \geq \epsilon\pi(s,u)) \leq 2\exp\left(-\frac{\epsilon^2 W\delta}{r_{max}\cdot(2+2\epsilon/3)}\right)$. Let $\lambda = \min(1, \sqrt{\frac{1}{\alpha W}})$, then we obtain:

$$Pr(|\tilde{\pi}(s,u) - \pi(s,u)| \geq \lambda\pi(s,u)) \leq 2\exp\left(-\frac{\lambda^2 W\delta}{r_{max}\cdot(2+2\lambda/3)}\right).$$

Since $W = \frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\delta}$, we derive $Pr(|\tilde{\pi}(s,u) - \pi(s,u)| \geq \lambda\pi(s,u)) \leq p_f^{\exp\left(-\frac{\lambda^2(2+2\epsilon/3)}{r_{max}\cdot\epsilon^2(2+2\lambda/3)}\right)}$.

Given that $\frac{\lambda^2(2+2\epsilon/3)}{r_{max}\cdot\epsilon^2(2+2\lambda/3)} > 1$, it follows that $Pr(|\tilde{\pi}(s,u) - \pi(s,u)| \geq \lambda\pi(s,u)) \leq p_f$. □

Proof for Theorem 4.8

Proof. The correctness of Algorithm 4 follows directly from [41] and Lemma 4.7. □

Proof for Theorem 4.9

Proof. The time complexity of Algorithm 4 when querying under adapted StackIndex$_\alpha$ consists of two parts: Forward-Push and further refinement. Forward-Push requires $O(\frac{\Delta_{out}}{\alpha r_{max}})$ [41], and refinement requires $O(\omega n)$. By setting r_{max} to $min(\sqrt{\frac{1}{\alpha\omega}}, 1)$. Query time complexity is $O(\min(\frac{\sqrt{\omega}n}{\sqrt{\alpha}}, \omega n))$. Combined with Theorem 4.2, the overall time complexity is $O(\omega\cdot(\mathrm{Tr}(I-\frac{P}{\beta+1})^{-1} - \mathrm{Tr}(I-\frac{P}{\bar{\beta}+1})^{-1} + n)))$. □

Proof for Theorem 5.1

Proof. In the graph $G\cup e$ obtained by adding the edge $e=(u,v)$, the probability that any stack element corresponding to u is v is $\frac{1}{d_u+1}$. This is exactly the case when an element of $Stack_u$ is chosen w.p. $\frac{1}{d_u+1}$. Using an geometric random variable to rapidly locate the first replacement does not affect this probability. For any other neighbor, the probability is $\frac{1}{d_u}\cdot\left(1-\frac{1}{d_u+1}\right) = \frac{1}{d_u+1}$ as expected. In contrast, in the graph $G\backslash e$ obtained by deleting the edge $e=(u,v)$, the probability that any stack element corresponding to u is v becomes 0, while for any other neighbor, the probability is $\frac{1}{d_u}+\frac{1}{d_u}\cdot\frac{1}{d_u-1} = \frac{1}{d_u-1}$ as expected. Thus, the sampling result is uniform. By Theorem 4.4, it follows that the spanning tree is uniform. Therefore, the new StackIndex satisfies the unbiased property. □

Proof for Theorem 5.2

Proof. Constructing a StackForest$_\alpha$ requires $O\left(\mathrm{Tr}\left(I-\frac{P}{1+\beta}\right)^{-1}\right)$ time. By previous discussion, we rebuild the index for less than $\sum_{u\in V}\frac{\frac{1}{\alpha}\pi(u,u)}{m}$ portion of the updates, in expectation. Note that $\sum_{u\in V}\frac{1}{\alpha}\pi(u,u) = \mathrm{Tr}\left(I-\frac{P}{1+\beta}\right)^{-1}$. Multiplying two terms yields the expected running time of $O(\omega\cdot\frac{\left(\mathrm{Tr}\left(I-\frac{P}{1+\beta}\right)^{-1}\right)^2}{m})$. □

A.2 Additional Experiments

Exp 7: Index building time across $\alpha\in(0,1)$. We conduct experiments to evaluate the index construction time for values of $\alpha\in(0,1)$. The parameter settings and comparison methods employed in these experiments are consistent with those described in the main text. The experimental results are shown in Fig. 13. From Fig. 13, we observe that for $\alpha>0.3$, the index construction times for all three methods are relatively small and exhibit minimal variation as α changes. Based on these observations, we can draw the following conclusions: (1) Building the meta-index for $\bar{\alpha}>0.3$ requires a similar amount of time as for $\bar{\alpha}=0.3$; (2) The adaptation time for $\alpha>0.3$ is small, as the differences in index construction time are negligible; (3) The primary advantage of StackIndex over SpeedPPR+ is evident in the range of $\alpha\in(0,0.3)$, as demonstrated in Section 6.

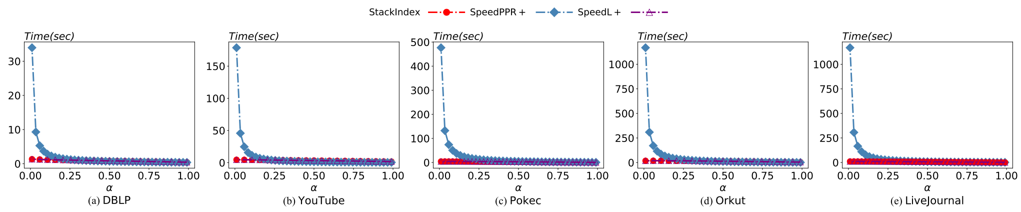


Fig. 13. Index construction time of different *index* methods - $\alpha \in (0, 1)$

Exp 8: Overall query performance with a relatively-large meta damping factor $\bar{\alpha} = 0.7$. In this section, we present the results of the overall query time experiment for the case where $\bar{\alpha} = 0.7$. The parameter settings and comparison methods used in these experiments are consistent with those outlined in the main text. As we vary $\bar{\alpha}$ from 0.3 to 0.7, the following observations can be made: (1) The relative performance in terms of query time across different methods remains consistent, with each query time showing a slight decrease as α increases. This is because querying with larger values of α generally takes less time. (2) The overall $L1$-error exhibits a slight decrease, while the maximum relative error increases. This can be attributed to the fact that for larger values of α, PPR values tend to concentrate more locally. As a result, there is a higher probability of encountering larger relative errors at nodes where the PPR value is small. On the other hand, this phenomenon reduces the number of nodes whose PPR values fall below the threshold in the relative error calculation, leading to a decrease in the $L1$-error.

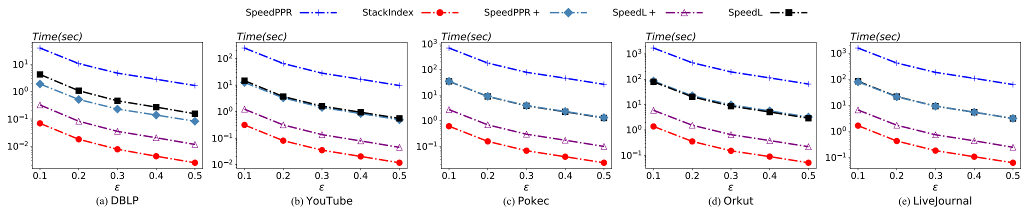


Fig. 14. Query processing time of different algorithms ($\bar{\alpha} = 0.7$)

Exp 9: Trace difference between various α. To evaluate the improvement of our α-adapting algorithm over SpeedL+, we calculate $\mathrm{Tr}(I - \frac{1}{\beta_2+1}P)^{-1}$ for different values of $\alpha = 0.3, 0.25, 0.15, 0.1, 0.05, 0.01$.

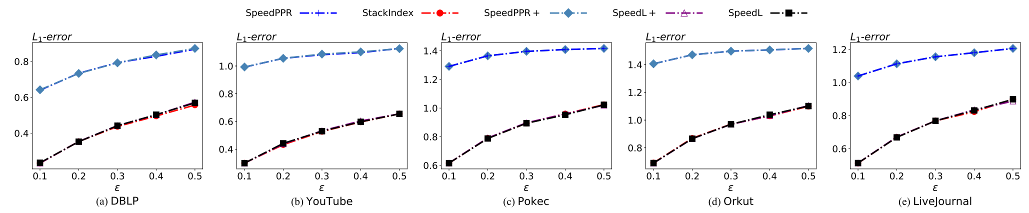


Fig. 15. Accuracy of different algorithms in terms of the $L1$-error metric ($\bar{\alpha} = 0.7$)

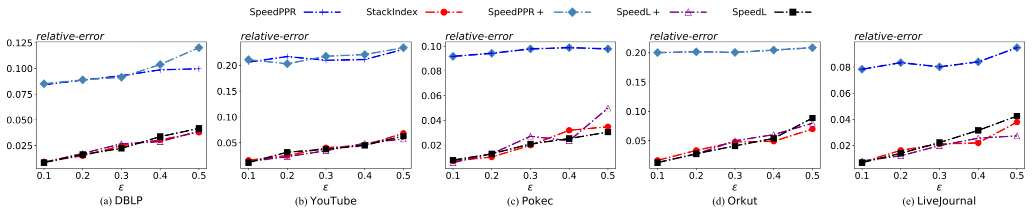


Fig. 16. Accuracy of different algorithms in terms of the maximum relative error metric ($\bar{\alpha} = 0.7$)

Table 6. Time complexity ratio between StackIndex and SpeedL+.

Dataset	(0.3, 0.25)	(0.3, 0.15)	(0.3, 0.1)	(0.3, 0.05)	(0.3, 0.01)
DBLP	0.04	0.14	0.19	0.25	0.33
Youtube	0.04	0.12	0.17	0.22	0.28
Pokec	0.03	0.09	0.12	0.14	0.16
Orkut	0.03	0.08	0.11	0.13	0.15
LiveJournal	0.03	0.10	0.13	0.17	0.20

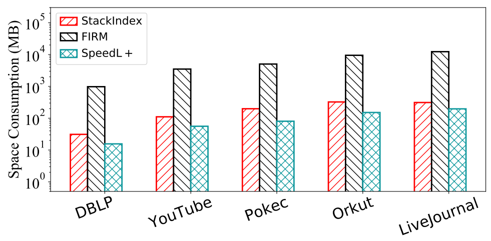


Fig. 17. Index space of different dynamic algorithms.

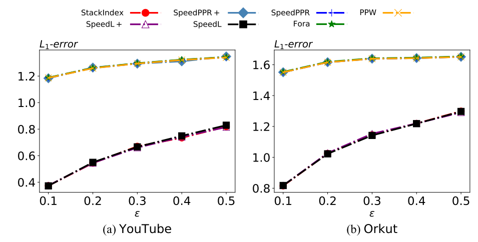


Fig. 18. Accuracy of different algorithms in terms of the L_1-error metric

These values represent the time complexity of SpeedL+, and the difference between them reflects the time complexity of StackIndex. The results are shown in Table 6. Each element, denoted as (α_1, α_2), is computed using the formula: $\frac{\mathrm{Tr}(I-\frac{1}{\beta_2+1}P)^{-1}-\mathrm{Tr}(I-\frac{1}{\beta_1+1}P)^{-1}}{\mathrm{Tr}(I-\frac{1}{\beta_2+1}P)^{-1}}$, where $\beta = \alpha/(1-\alpha)$. This formula represents the time complexity ratio between StackIndex and SpeedL+. The result indicates that StackIndex significantly reduces the time complexity compared to SpeedL+. Specifically, we observe that StackIndex only requires a fraction of the time taken by SpeedL+, leading to a substantial improvement in efficiency. For example, in Pokec and Orkut, StackIndex only takes less than 20% ratio of the time even when adapting α from 0.3 to 0.01.

Exp 10. Space consumption of StackIndex **in dynamic graphs.** We evaluate the space consumption of different indexing algorithms in dynamic graphs. The key difference is that, in dynamic graphs, we store the Stack structure within StackIndex. We compare StackIndex with SpeedL+ and FIRM (the dynamic version of SpeedPPR+) in the main text. The results are shown in Fig. 17. We observe the following: (1) The space increases due to the Stack structure does not exceed the magnitude of the original size. For example, StackIndex in a static graph requires 201 MB, while in a dynamic graph it requires 324 MB. Since storing Orkut itself requires 895 MB, this space increase is manageable. (2) The space consumption of StackIndex is significantly less than that of FIRM. While FIRM requires more than 1 GB in the smallest dataset DBLP, StackIndex requires only 324 MB in the largest dataset Orkut. These results demonstrate that our index updating approach is highly space efficient.

Exp 11: Overall query performance ($L1$-error). We estimate the $L1$-error of different algorithms over queries under different parameter ϵ, as in Exp 1. The result is shown in Fig. 18. As expected, the $L1$-error of StackIndex, SpeedL, and SpeedL+ are very close. This is because all three algorithms are based on the random spanning forest sampling-based method. Additionally, StackIndex, SpeedL, and SpeedL+ exhibit significantly higher accuracy compared to FORA, PPW, SpeedPPR and SpeedPPR+. These results strengthen that our StackIndex outperforms existing *index* and *online* methods in terms of both query time and accuracy.

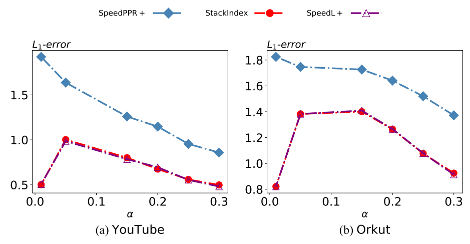


Fig. 19. Comparison of the L_1-error of different indexing algorithms with varying α

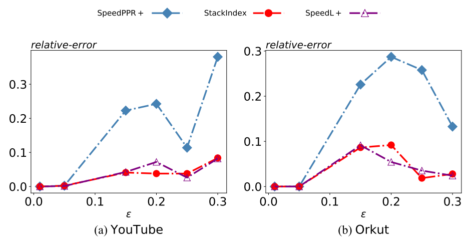
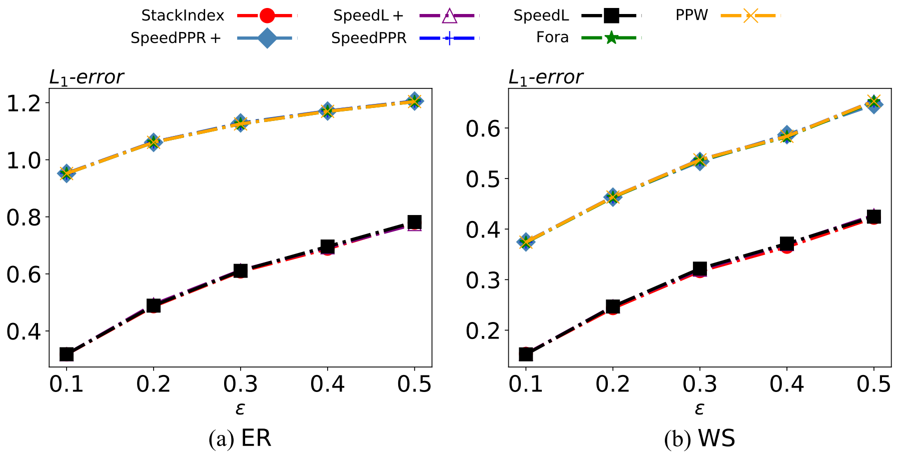


Fig. 20. Comparison of the maximum relative error of different indexing algorithms with varying α



(a) $L1$-error with varying ϵ

Fig. 21. $L1$-error on synthetic datasets when varying ϵ

Exp 12: Accuracy of indexing algorithms with varying α. We estimate $L1$-error and maximum relative error of different algorithms during transformation, as in Exp 3. The result is shown in Fig. 19 and 20. As observed, for $L1$-error and maximum relative error, StackIndex exhibits comparable accuracy to SpeedL+, both significantly outperform SpeedPPR+ which are consistent with our previous experimental results. These findings further confirm the high accuracy performance of the proposed solution.

Exp 13: Overall query performance on synthetic graphs ($L1$-error). We estimate $L1$-error of different algorithms over queries on synthetic graphs, as in Exp 5. The result is shown in Fig. 21. This result is consist with those on real-world graphs, demonstrating that StackIndex outperforms other algorithms in terms of $L1$-error.

Exp 14: Overall query performance under $\bar{\alpha} = 0.9$ ($L1$-error and maximum relative error) When changing the damping factor to $\bar{\alpha} = 0.9$, We estimate the $L1$-error and maximum relative error of different algorithms over queries under different parameter ϵ, as in Exp 6. The result is shown in Fig. 22 and 23. As expected, the accuracy of StackIndex is comparable to SpeedL+ and significantly higher than other baseline methods. These findings align with the observations in Exp 11, indicating that our method maintains strong performance under different meta-damping factor settings.

Exp 15: Query time of different algorithms varying α. To further analyze the overall query performance of different algorithms, Figure 24 illustrates the step-by-step query time for each α,

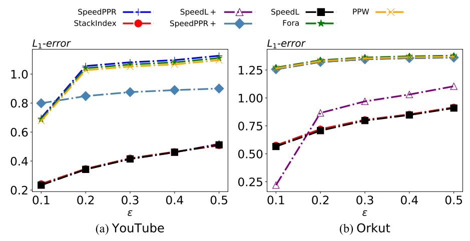


Fig. 22. Accuracy of different algorithms in terms of the L_1-error metric ($\bar{\alpha} = 0.9$)

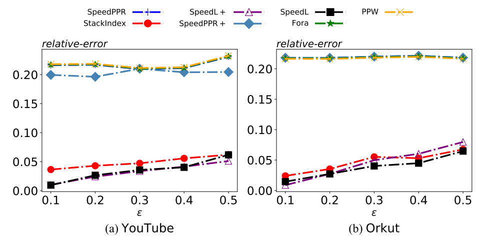


Fig. 23. Accuracy of different algorithms in terms of the maximum relative error metric ($\bar{\alpha} = 0.9$)

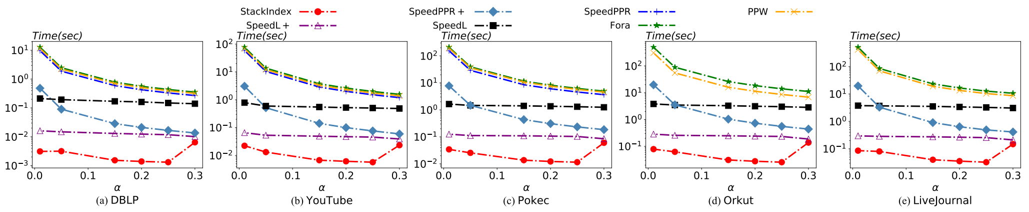


Fig. 24. Comparison of the indexing (transforming) time of different algorithms with varying α

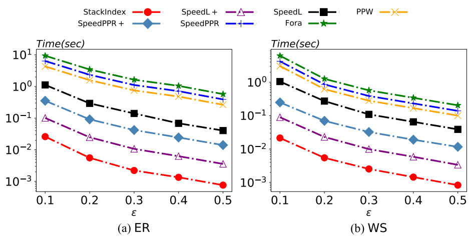


Fig. 25. Overall performance on synthetic datasets: Running time with varying ϵ

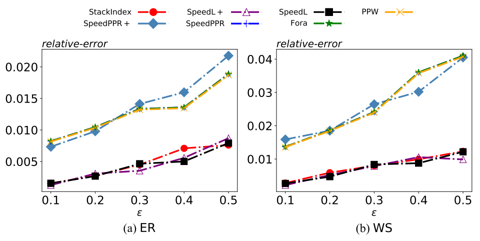


Fig. 26. Overall performance on synthetic datasets: Maximum error with varying ϵ

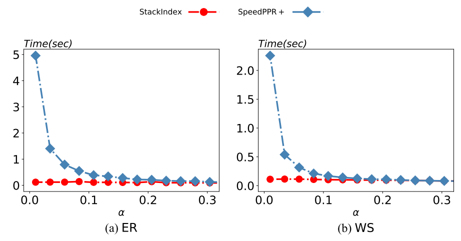


Fig. 27. Index building time of different algorithms on synthetic datasets

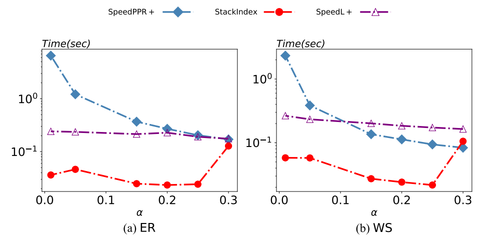


Fig. 28. Runtime with varying α on synthetic datasets

including both index-based and index-free methods. The results show that, except for the initial query with $\alpha = 0.3$, StackIndex significantly outperforms other methods in terms of query time. This advantage is particularly pronounced for smaller α values. Additionally, since the comparison is based on average query time and index-based methods can reuse their indices, their query time for each step is substantially lower than that of the corresponding index-free methods.

Exp 16: Overall query performance on synthetic graphs. Figure 25, 21 and 26 illustrates the relationship between α and the query time on synthetic graphs, as in Exp 1. The result is consisted with that on real-world graphs. StackIndex significantly outperforms other methods in

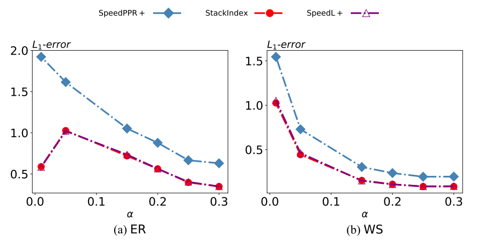


Fig. 29. L_1-error with varying α on synthetic datasets

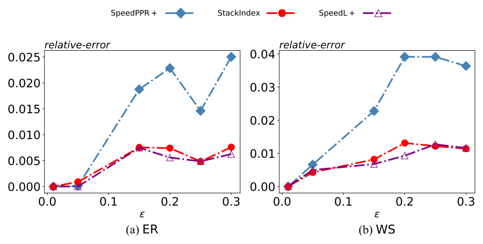


Fig. 30. Maximum relative error with varying α on synthetic datasets

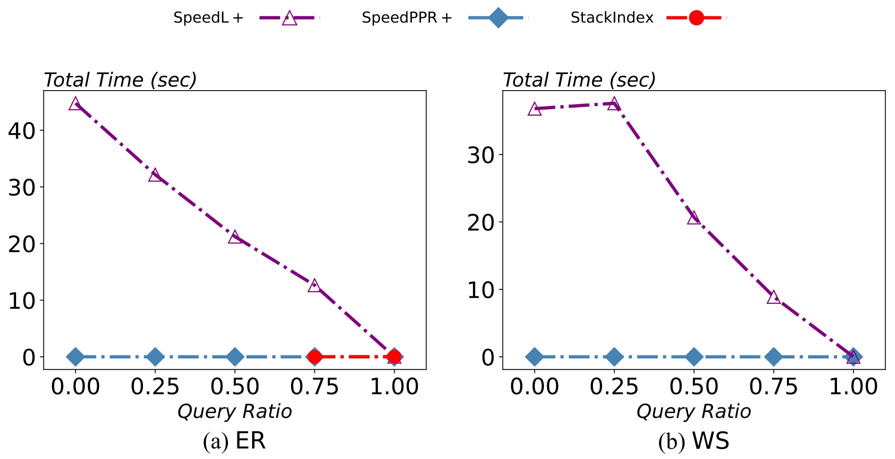


Fig. 31. Index updating performance on the dynamic synthetic graph under different workloads

term of query time, and StackIndex exhibits comparable accuracy to SpeedL+, both significantly outperform SpeedPPR+.

Exp 17: Index building time on synthetic graphs. Figure 27 illustrates the relationship between α and the index building time on synthetic graphs, as in Exp 2. The result is consisted with that on real-world graphs. The index construction time for StackIndex and SpeedL+ is nearly identical, and the index construction time for both StackIndex and SpeedL+ is robust with respect to α. When α is small, the index construction time of StackIndex and SpeedL+ is significantly lower than that of SpeedPPR+, and comparable when $\alpha \geq 0.2$.

Exp 18: Runtime and accuracy with varying α on synthetic datasets. Figure 28, 29 and 30 illustrates the runtime and accuracy with varying α on synthetic datasets on synthetic graphs, as in Exp 3. The results are consisted with those on real-world graphs. StackIndex significantly outperforms other methods in term of runtime, and StackIndex exhibits comparable accuracy to SpeedL+, both significantly outperform SpeedPPR+.

Exp 19: Index updating performance on synthetic graphs. Figure 31 illustrates index updating performance on the dynamic synthetic graphs, as in Exp 4. The result is consisted with that on real-world graphs. StackIndex significantly outperforms SpeedL+ in term of updating time, while remains manageable.

Table 7. **Performance of node classification by PPR-based scalable GNN with Varying α. The highest F1-score in each column is marked in red.**

Dataset	Cora	Pubmed	Citeseer	PPI	Yelp
α	**F1-score**				
0.01	0.786	0.746	0.649	0.463	0.516
0.05	0.832	0.806	0.670	0.791	0.577
0.1	0.833	0.803	0.694	0.905	0.592
0.2	0.834	0.793	0.710	0.955	0.606
0.3	0.826	0.796	0.734	0.959	0.618
0.4	0.796	0.773	0.746	0.953	0.625
0.5	0.760	0.783	0.798	0.941	0.631
0.6	0.746	0.762	0.811	0.911	0.635
0.7	0.720	0.755	0.832	0.845	0.638
0.8	0.700	0.750	0.831	0.724	0.640
0.9	0.675	0.730	0.829	0.596	0.642

A.3 Full Table in case study

In this experiment, we illustrate the full result of our two case studies. Table 7 and Table 8 correspond to Application I and II, respectively. Notably, the F1-score achieved with the optimal α values is significantly higher than those with other α values (for exmaple, in Cora, F1-score is 0.675 when $\alpha = 0.2$, 0.834 when $\alpha = 0.2$, making a difference of 0.15 with different α), underscoring the effectiveness of our approach in improving clustering quality. Similar result can also be found in conductance of local graph clustering, for example, in DBLP, conductance is 0.2243 when $\alpha = 0.01$, 0.3688 when $\alpha = 0.9$, making a difference of 0.1445 with different α. This indicates that our approach can effectively improve the clustering quality by selecting the optimal α value.

Table 8. **Performance of ppr-based local graph clustering with Varying α. The lowest conductance in each column is marked in red.**

Dataset	DBLP			Youtube			LiveJournal			Orkut			Amazon		
α	**Conductance**	**F1-score**	**Purity**	**Conductance**	**F1-score**	**Purity**	**Conductance**	**F1-score**	**Purity**	**Conductance**	**F1-score**	**Purity**	**Conductance**	**F1-score**	**Purity**
0.01	0.2243	0.4972	0.3066	0.7478	0.4120	0.5010	0.8798	0.0192	0.0714	0.7677	0.4012	0.5231	0.5176	0.8486	0.9386
0.05	0.2255	0.4973	0.3063	0.7478	0.4135	0.5021	0.8746	0.0182	0.0500	0.7720	0.4078	0.5289	0.5176	0.8486	0.9386
0.1	0.2259	0.4975	0.3077	0.7476	0.4152	0.8820	0.8820	0.0182	0.0500	0.7734	0.4123	0.6136	0.6136	0.8356	0.9187
0.2	0.2260	0.4970	0.3054	0.7482	0.4170	0.5045	0.7911	0.2948	0.3485	0.7771	0.4178	0.5356	0.5302	0.7740	0.9173
0.3	0.2265	0.4977	0.3034	0.7490	0.4185	0.5056	0.8880	0.0192	0.0714	0.7801	0.5312	0.5389	0.5312	0.7684	0.9102
0.4	0.2267	0.4974	0.3039	0.7501	0.4198	0.5067	0.8880	0.0192	0.0714	0.7819	0.4289	0.5412	0.5312	0.7684	0.9102
0.5	0.2271	0.4994	0.3061	0.7512	0.4212	0.5078	0.8784	0.0661	0.1290	0.7828	0.4345	0.5434	0.5316	0.7743	0.9193
0.6	0.2273	0.4982	0.3050	0.7523	0.4225	0.5089	0.8792	0.0667	0.1333	0.7857	0.4398	0.5456	0.5316	0.7743	0.9173
0.7	0.2284	0.4979	0.3056	0.7534	0.4238	0.5101	0.8814	0.0672	0.1379	0.7891	0.4456	0.5478	0.4322	0.7662	0.9231
0.8	0.2291	0.4987	0.3051	0.7545	0.4251	0.5112	0.8860	0.0833	0.1667	0.7918	0.4512	0.5490	0.4323	0.7636	0.9253
0.9	0.3688	0.4855	0.6185	0.7556	0.4264	0.5123	0.8860	0.0833	0.1667	0.7958	0.4567	0.5501	04330	0.7452	0.9358

Received Jan 2025; revised May 2025; accepted May 2025