HubPPR: Effective Indexing for Approximate Personalized PageRank

Sibo Wang[†], Youze Tang[†], Xiaokui Xiao[†], Yin Yang[‡], Zengxiang Li[§]
[†]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[‡]College of Science and Engineering, Hamad Bin Khalifa University, Qatar
[§]Institute of High Performance Computing, A*STAR, Singapore
[†]{wang0759, yztang, xkxiao}@ntu.edu.sg, [‡]yyang@qf.org.qa, [§]liz@ihpc.a-star.edu.sg

ABSTRACT

Personalized PageRank (PPR) computation is a fundamental operation in web search, social networks, and graph analysis. Given a graph G, a source s, and a target t, the PPR query $\pi(s,t)$ returns the probability that a random walk on G starting from s terminates at t. Unlike global PageRank which can be effectively pre-computed and materialized, the PPR result depends on both the source and the target, rendering results materialization infeasible for large graphs. Existing indexing techniques have rather limited effectiveness; in fact, the current state-of-the-art solution, *BiPPR*, answers individual PPR queries without pre-computation or indexing, and yet it outperforms all previous index-based solutions.

Motivated by this, we propose *HubPPR*, an effective indexing scheme for PPR computation with controllable tradeoffs for accuracy, query time, and memory consumption. The main idea is to pre-compute and index auxiliary information for selected hub nodes that are often involved in PPR processing. Going one step further, we extend HubPPR to answer top-k PPR queries, which returns the k nodes with the highest PPR values with respect to a source s, among a given set T of target nodes. Extensive experiments demonstrate that compared to the current best solution *BiPPR*, *HubPPR* achieves up to 10x and 220x speedup for PPR and top-k PPR processing, respectively, with moderate memory consumption. Notably, with a single commodity server, *HubPPR* answers a top-k PPR query in seconds on graphs with billions of edges, with high accuracy and strong result quality guarantees.

1. INTRODUCTION

Personalized PageRank (PPR) [25] is a fundamental metric that measures the relative importance of nodes in a graph from a particular user's point of view. For instance, search engines use PPR to rank web pages for a user with known preferences [25]. Microblogging sites such as Twitter use PPR to suggest to a user other accounts that s/he might want to follow [16]. Additionally, PPR can be used for predicting and recommending links in a social network [5], and analyzing the relationship between different nodes on large graph data such as protein networks [17].

Specifically, given a graph G, a source node s, and a target node t, the PPR $\pi(s,t)$ of t with respect to s is defined as the probability that a random walk on G starting from s terminates at t. For example, on a social networking site where nodes correspond to user profiles, $\pi(s,t)$ measures the importance of user t from user s's perspective; hence, if $\pi(s,t)$ is high and s and t are not already connected, the social networking site might want to recommend t to s. Another important variant is top-k PPR, in which there are a set T of target nodes, and the goal is to identify nodes among T with the highest PPR values with respect to s. This can be applied, for example, to the selection of the top web documents among those retrieved through a keyword query [20].

As we explain in Section 2, exact PPR computation incurs enormous costs for large graphs; hence, the majority of existing work focuses on approximate PPR computation. Meanwhile, since the PPR result depends on both the source and target nodes, it is prohibitively expensive to materialize the results of all possible PPR queries for a large graph. Accelerating PPR processing through indexing is also a major challenge, and the effectiveness of previous indices has been shown to be rather limited [8, 9]. In fact, the current state-of-the-art solution for PPR processing is *BiPPR* [20], which answers each PPR query individually without any pre-computation or indexing. It is mentioned in [20] that *BiPPR* could benefit from materialization of partial results. However, as we explain in Section 2.2, doing so is not practical for large graphs due to colossal space consumption.

Regarding top-k PPR processing, existing methods are either not scalable, or fail to provide formal guarantees on result quality. Further, as we describe in Section 2, similar to the case of PPR computation, for the top-k PPR query, an adaptation of *BiPPR* remains the state of the art, which processes queries on the fly without indices. In other words, to our knowledge no effective indexing scheme exists for top-k PPR processing that can accelerate or outperform *BiPPR* without sacrificing the query accuracy.

Motivated by this, we propose *HubPPR*, an effective index-based solution for PPR and top-k PPR processing. Particularly for top-k PPR, *HubPPR* contains a novel processing framework that achieves rigorous guarantees on result quality; at the same time, it is faster and more scalable than *BiPPR* even without using an index. *HubPPR* further accelerates PPR and top-k PPR through an *elastic hub index (EHI)* that (i) adapts well to the amount of available memory and (ii) can be built by multiple machines in parallel. These features render *HubPPR* a good fit for modern cloud computing environments.

The EHI contains pre-computed aggregate random walk results from a selected set of hub nodes which are likely to be involved in PPR computations. Figure 1 shows an example graph with two

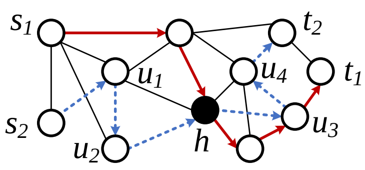


Figure 1: Example of a hub node and random walks

random walk trajectories, one from node s_1 to t_1, and another from s_2 to t_2. Node h appears in both trajectories. As we elaborate later, if we select h as a hub, precompute and index random walk destinations starting from h in the *forward oracle* of the EHI, then, random walks, e.g., the two starting from s_1 and s_2, can terminate as soon as they reach h, and *HubPPR* determines their final destinations using the forward oracle. Similarly, the EHI also contains a more sophisticated *backward oracle* storing additional aggregate information about the hubs. To build an effective index with limited memory space, *HubPPR* addresses several important technical challenges, including choosing the hubs, storing and compressing their associated aggregates, and ensuring that the guarantees on result quality are satisfied.

Extensive experiments using real data demonstrate that with moderate memory consumption, *HubPPR* achieves up to 10x (resp. 220x) reduction in query time for approximate PPR (resp. top-k PPR) processing compared to the current best method *BiPPR*. Notably, using a single commodity server, *HubPPR* answers an approximate top-k PPR query in seconds on a billion-edge graph, with high result quality.

2. PRELIMINARIES

Section 2.1 provides the necessary background on PPR and top-k PPR. Section 2.2 presents our main competitor *BiPPR*, the current state-of-the-art approach to PPR and top-k PPR processing. Table 1 lists the notations that will be frequently used in the paper.

2.1 Problem Definition

Personalized PageRank. Given a graph $G = (V, E)$ where V (resp. E) is the set of nodes (resp. edges) in G, a source node $s \in V$, a target node $t \in V$, and a probability α, the *personalized PageRank (PPR)* $\pi(s,t)$ of t with respect to s is defined as the probability that a random walk on G from s terminates at t. Accordingly, the PPR values for all nodes in the graph sum up to 1. In particular, in each step of the random walk, let v be the current node; with probability α, the random walk terminates at v; with probability $1-\alpha$, it picks an out edge $(v,w) \in E$ of v uniformly at random and follows this edge to reach node w. The random walk eventually terminates at a node in V, which we call the destination of the walk.

The exact PPR values for all nodes in G with respect to a particular source node s can be computed by the *power iterations* method described in the original paper on PPR [25], which remains the basis of modern exact PPR processing methods [22]. In a nutshell, power iterations can be understood as solving a matrix equation. Specifically, let n denote the number of nodes in G, and $A \in \{0,1\}^{n\times n}$ be the adjacency matrix of G. We define a diagonal matrix $D \in R^{n\times n}$ in which each element on its main diagonal corresponds to a node v, and its value is the out degree of v. Then, we have the following equation:

$$\boldsymbol{\pi_s} = \alpha \cdot \boldsymbol{e_s} + (1-\alpha)\cdot \boldsymbol{\pi_s} \cdot D^{-1}A. \quad (1)$$

where $\boldsymbol{e_s}$ is the identity vector of s, and $\boldsymbol{\pi_s}$ is the PPR vector for node s that stores PPR of all nodes in V with respect to s. Solving the above equation involves multiplying matrices of size n by n, which takes $O(n^c)$ time, where the current lowest value for constant c is $c \approx 2.37$ [14]. This is immensely expensive for large graphs. Another issue is that storing the adjacency matrix A takes $O(n^2)$ space, which is prohibitively expensive for a graph with a large number of nodes. Although there exist solutions for representing A as a sparse matrix, e.g., [13], such methods increase the cost of matrix multiplication, exacerbating the problem of high query costs. Finally, since there are $O(n^2)$ possible PPR queries with different source/target nodes, materializing the results for all of them is clearly infeasible for large graphs.

Table 1: Frequently used notations.

Notation	Description
G=(V,E)	Input graph, its node set and edge set
n, m	The number of nodes and edges in G, respectively
$\pi(s,t)$	Exact result of a PPR query with source s and target t
α	Probability for a random walk to terminate at each step
δ, ϵ, p_f	Parameters for the result quality guarantee of an approximate PPR algorithm, described in Definition 1
$r(v,t)$	The residue of v during backward search from t
$\pi^{-1}(v,t)$	The reserve of v during backward search from t
r_{max}	Residue threshold for backward propagation
$\mathcal{F}, \mathcal{B}$	Forward and backward oracles, respectively
ω	Number of random walks during forward search

Approximate PPR. Due to the high costs for computing the exact PPR, most existing work focuses on approximate PPR computation with result accuracy guarantees. It has been shown that when the PPR value is small, it is difficult to obtain an approximation bound on accuracy [20, 21]. Meanwhile, large PPR results are usually more important in many applications such as search result ranking [25] and link prediction [5]. Hence, existing work focuses on providing accuracy guarantees for PPR results that are not too small. A popular definition for approximate PPR is as follows.

DEFINITION 1. *Given a PPR query $\pi(s,t)$, a result threshold δ, an approximation ratio ϵ, and a probability α, an approximate PPR algorithm guarantees that when $\pi(s,t) > \delta$, with probability at least $1-p_f$, we have:*

$$|\hat{\pi}(s,t) - \pi(s,t)| \le \varepsilon \cdot \pi(s,t), \quad (2)$$

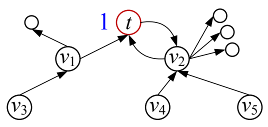
where $\hat{\pi}(s,t)$ is the output of the approximate PPR algorithm. □

A common choice for δ is $O(1/n)$, where n is the number of nodes in the graph. The intuition is that if every node has the same PPR value, then this value is $1/n$, since the PPR values for all nodes sum up to 1; hence, by setting δ to $O(1/n)$, the approximation bound focuses on nodes with above-average PPR values.

Top-*k* PPR. As mentioned in Section 1, top-k PPR concerns the selection of k nodes with the highest PPR values among a given set T of target nodes. As shown in [20], finding the exact answer to a top-k query also incurs high costs, especially when the target set T contains numerous nodes, e.g., web pages matching a popular keyword. Meanwhile, materializing all possible top-k PPR results is inapplicable, since the result depends on the target set T, whose number can be exponential. Unlike the case for PPR queries, to our knowledge no existing work provides any formal guarantee on result quality for top-k PPR processing. In Section 4, we formally define the problem of approximate top-k PPR, and present an efficient solution with rigorous guarantees on result quality.

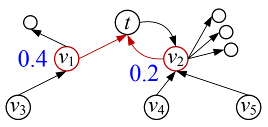
2.2 BiPPR

BiPPR [20] processes a PPR query through a bi-directional search on the input graph. First, *BiPPR* performs a backward search using a backward propagation algorithm originally described in [1].



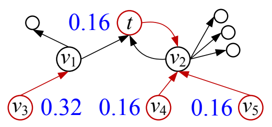
node	t	v_1	v_2	v_3	v_4	v_5
reserve	0	0	0	0	0	0

(a) Step 1.



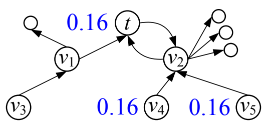
node	t	v_1	v_2	v_3	v_4	v_5
reserve	**0.2**	0	0	0	0	0

(b) Step 2.



node	t	v_1	v_2	v_3	v_4	v_5
reserve	0.2	**0.08**	**0.04**	0	0	0

(c) Step 3.



node	t	v_1	v_2	v_3	v_4	v_5
reserve	0.2	0.08	0.04	**0.064**	0	0

(d) Step 4.

Figure 2: Example of backward search.

Then, it performs forward searches based on Monte Carlo simulations [24]. Finally, it combines the results for both search directions, and estimates the PPR result. In the following, we explain the forward and backward search of *BiPPR*, the combination of the results, the extension to top-k PPR queries, as well as the possibility of accelerating *BiPPR* through materializing search results.

Forward search. The forward search performs ω random walks starting from the source node s. Let h_v be the number of random walks that terminate at each node $v \in V$. Then, the forward search estimates the PPR $\pi_f(s, v) = h_v/\omega$. According to properties of Monte Carlo simulations [24], $\pi_f(s, v)$ is an unbiased estimate of the exact PPR $\pi(s, t)$. Meanwhile, using the Chernoff bound [24], we can prove that π_f satisfies Inequality 2 when $\omega \geq \frac{3\log(2/p_f)}{\epsilon^2\delta}$, where δ, ϵ, p_f are result threshold, the approximation ratio and failure probability, respectively, as in Definition 1. Note that when $\delta = O(1/n)$ (as explained in Section 2), the number of random walks is $O(\frac{n \cdot log(1/p_f)}{\epsilon^2})$, which is costly for large graphs.

Backward search. The backward search in *BiPPR* starts from the target node t and propagates information along the reverse direction of the edges. The search iteratively updates two properties for each node v: its residue $r(v, t)$, and reserve $\pi^{\dashv}(v, t)$ [1]. The former represents information to be propagated to other nodes, and the latter denotes the estimated PPR value of target t with respect to node v. The search starts from the state in which *(i)* every node in the graph has zero reserve, *(ii)* every node, except for the target node t, has zero residue as well and *(iii)* t has a residue of $r(t, t) = 1$. In each iteration, residues are propagated between nodes and converted to reserves. The goal is to eventually deplete the residue for every node, i.e., $r(v, t) = 0$ for all $v \in V$, at which point the search completes. It has been proven that if we follow the propagation rules in [1] that will be explained shortly, then after the search finishes, the reserve for each node v equals the exact PPR of t with respect to v, i.e., $\pi^{\dashv}(v, t) = \pi(v, t)$ [1]. *BiPPR* does not complete the backward search, which is expensive; instead, it sets a maximum residue r_{max}, and terminates the search as soon as every node v satisfies $r(v, t) < r_{max}$ [20]. Next we clarify the residue propagation and conversion rules. For each iteration of the backward search, for every $v \in V$ with sufficient residue $r(v, t) > r_{max}$, *BiPPR* converts α portion of the residue to its reserve, i.e., $\pi^{\dashv}(v, t)$ is incremented by $\alpha \cdot r(v, t)$. Then, the remaining residue is propagated along the reverse direction of in-edges of v. In particular, for each edge $(u, v) \in E$, let $d_{out}(u)$ be the out degree of node u; *BiPPR* increments its residue $r(u, t)$ by $\frac{(1-\alpha)\cdot r(v,t)}{d_{out}(u)}$. After finishing the propagation along all in-edges of v, *BiPPR* sets $r(v, t)$ to zero. In case v does not have any in-edge, *BiPPR* will not propagate information and directly set $r(v, t)$ to zero. We demonstrate how the backward search works with the following example.

Example 1. Consider graph G in Figure 2(a). Assume that t is the target node, $\alpha = 0.2$, and $r_{max} = 0.18$. The backward search starts with t with $r(t, t) = 1$, and $r(v, t) = 0$ for every remaining node v. We illustrate the non-zero residues alongside the corresponding nodes, and we list the reserve of each node in the table below each figure.

In the first iteration (i.e., Figure 2(b)), t has residue $r(t, t) > r_{max}$. Therefore, it converts $\alpha \cdot r_{max}(t, t)$ to its reserve, i.e., $\pi^{\dashv}(t, t) = 0.2$. Afterwards, it propagates the remaining residue to its in-neighbors, i.e., v_1 and v_2. As v_1 has two out-neighbors, $r(v_1, t)$ is incremented by $0.8/2 = 0.4$. Similarly, $r(v_2, t)$ is incremented by 0.2. Afterwards, as v_1 has residue larger than r_{max}, it converts $0.2 * 0.4 = 0.08$ residue to its reserve. It then propagates the remaining residue to its only in-neighbor (i.e., v_3), which increases $r(v_3, t)$ to 0.32. Similary, as v_2 also has residue larger than r_{max}, it converts 0.2 portion of its residue to its reserve, and then propagates the remaining residue to its in-neighbors. The results after the two propagations are shown in Figure 2(c).

Observe that v_3 still has residue larger than r_{max}. As v_3 has no in-neighbors, it simply increments its reserve by $0.32 * 0.2 = 0.064$, and sets its residue $r(v_3, t) = 0$. At this stage, every node v has $r(v, t) < r_{max}$. Hence, the backward propagation terminates. Figure 2(d) shows the final results. □

Combining forward and backward search results. As mentioned earlier, *BiPPR* first performs a backward search with residue threshold r_{max}, which obtains the reserve $\pi^{\dashv}(v, t)$ and residue $r(v, t) < r_{max}$ for each node v. Regarding $\pi^{\dashv}(v, t)$ and $r(v, t)$, it is proven in [20] that the following equation holds.

$$\pi(s, t) = \pi^{\dashv}(s, t) + \sum\nolimits_{v \in V} \pi(s, v) \cdot r(v, t). \tag{3}$$

Based on Equation 3, *BiPPR* answers the PPR query using the following equation.

$$\hat{\pi}(s, t) = \pi^{\dashv}(s, t) + \sum\nolimits_{v \in V} \pi_f(s, v) \cdot r(v, t). \tag{4}$$

It is proven in [20] that (i) *BiPPR* satisfies Definition 1 with $\omega = O\left(\frac{r_{max}}{\varepsilon^2\delta}\log\frac{1}{p_f}\right)$ random walks during the forward search, and (ii) if the target node t is chosen uniformly at random, then the average running time of the backward search is $O(\frac{m}{n\alpha r_{max}})$, where n and m are the number of nodes and number of edges in the graph, respectively[1]. By combing the costs of both the forward search and backward search, the average query time of *BiPPR* is $O\left(\frac{m}{n\alpha r_{max}} + \frac{r_{max}}{\alpha\varepsilon^2\delta}\log\frac{1}{p_f}\right)$. Lofgren et al. [20] recommends setting r_{max} to $\sqrt{\frac{m\cdot\varepsilon^2\delta}{n\log(1/p_f)}}$, which leads to an average time complexity of $O\left(\frac{1}{\alpha\varepsilon}\sqrt{\frac{m}{n\delta}}\log\frac{1}{p_f}\right)$.

Materialization of search results. Ref. [20] mentions that *BiPPR* could be accelerated by pre-computing and materializing the bi-directional search results. Specifically, for forward search, *BiPPR* could simply materialize the final destinations of the ω random walks for each node $v \in V$. The space consumption, however, is $O\left(\frac{1}{\varepsilon}\sqrt{\frac{mn}{\delta}}\log\frac{1}{p_f}\right)$. As $\delta = O(1/n)$, the space complexity is hence $O\left(\frac{n}{\varepsilon}\sqrt{m\log\frac{1}{p_f}}\right)$, which is impractical for large graphs.

Regarding backward search, *BiPPR* could perform a backward search starting from each node t during pre-processing, and materialize the residues and reserves of all nodes. According to [20],

[1]The worst-case running time of the backward search is $\Theta(n)$.

the total space consumption is $O\left(\frac{m}{r_{max}}\right) = \left(\frac{1}{\varepsilon}\sqrt{\frac{mn}{\delta}\log\frac{1}{p_f}}\right)$. When $\delta = O(1/n)$, the space complexity is $O\left(\frac{n}{\varepsilon}\sqrt{m\log\frac{1}{p_f}}\right)$, which is also infeasible for large graphs.

Top-k PPR processing. Observe that *BiPPR* can also be applied to top-k PPR in a straightforward manner: given a target set T, we can employ forward and backward searches to estimate the PPR score of each node in T, and return the k nodes with the highest scores. This approach, however, is rather inefficient as it requires performing $|T|$ PPR queries. To improve efficiency, Lofgren et al. [20] consider the restricted case when T is selected from a small number of possible choices, and propose preprocessing techniques leveraging the knowledge of T to accelerate queries. Nevertheless, those techniques are inapplicable when T is arbitrary and is given only at query time. Therefore, efficient processing of top-k PPR queries for arbitrary target set T remains a challenging problem.

3. HUBPPR

Similar to *BiPPR* described in Section 2.2, *HubPPR* performs forward and backward searches, and combines their results to answer a PPR query. The main distinction of *HubPPR* is that it performs forward (resp. backward) search with the help of a pre-computed index structure called the *forward oracle* (resp. *backward oracle*). To facilitate fast processing of PPR queries, we assume that the index resides in main memory, which has limited capacity. This section focuses on the *HubPPR* algorithm; the data structures for forward and backward oracles are described later in Section 5. In the following, Sections 3.1 and 3.2 describe index-based forward and backward searches in *HubPPR*, respectively. Section 3.3 presents the complete *HubPPR* algorithm, proves its correctness, and analyzes its time complexity.

3.1 Forward Search

We first focus on the forward search. Recall from Section 2.2 that the forward search performs a number of random walks. *HubPPR* accelerates a random walk based on its memorylessness property, stated as follows.

LEMMA 1 (MEMORYLESSNESS OF A RANDOM WALK). *Let $v \in V$ be any node reachable from the source node s, and B_v denote the event that a random walk P starting from s reaches v. Then, for any node t, $Pr[P$ terminates at $t \mid B_v] = \pi(v,t)$.* □

Our forward oracle is motivated by Lemma 1, defined as follows.

DEFINITION 2 (FORWARD ORACLE). *A forward oracle $\mathcal{F}$ is a data structure that for any input node $v \in V$, $\mathcal{F}$ either returns NULL, or the destination node $w \in V$ of a random walk starting from s that reaches v.* □

Note that in the above definition, it is possible that $\mathcal{F}$ returns different results for different probes with the same input node v, which correspond to different random walk destinations. Given a forward oracle $\mathcal{F}$, *HubPPR* accelerates the forward search as follows. When performing a random walk during forward search, whenever *HubPPR* reaches a node v, it probes $\mathcal{F}$ with v. If $\mathcal{F}$ returns NULL, *HubPPR* continues the random walk. Otherwise, i.e., $\mathcal{F}$ returns a node w, *HubPPR* terminates the random walk and returns w as its destination.

EXAMPLE 2. Assume that we perform a forward search from node s_2 in Figure 1, and the sequence of nodes to be traversed in the random walk is $s_2, u_1, u_2, h, u_3, u_4, t_2$ (as illustrated in the dashed line in Figure 1). Then, the forward search first probes $\mathcal{F}$ with s_2. If $\mathcal{F}$ returns NULL, the forward search would continue the random walk from s_2 and jumps to u_1, after which it probes $\mathcal{F}$ with u_2. If $\mathcal{F}$ still returns NULL, the forward search jumps from u_2 to h, and probes $\mathcal{F}$ with h. Assume that $\mathcal{F}$ returns t_1. Then, the random walk terminates immediately and returns t_1 as the destination. □

Challenges in designing $\mathcal{F}$. There are three main requirements in the design of $\mathcal{F}$. First, $\mathcal{F}$ must be effective in reducing random walk costs; in particular, it should minimize NULL responses. Second, $\mathcal{F}$ should be *space efficient* as it resides in memory. Third, $\mathcal{F}$ should also be *time efficient* in responding to probes; in particular, $\mathcal{F}$ must process each probe in constant time (in order not to increase the time complexity of forward search), and the constant must be small (in order not to defeat the purpose of indexing).

A naive solution is to materialize ω random walk destinations for each node, which incurs prohibitive memory consumption, as explained in Section 2.2. To conserve space, we can store random walks destinations *selectively*, and in a *compressed* format. On the other hand, selective storage may compromise indexing efficiency, and compression may lead to probing overhead. What should we materialize to best utilize the available memory? Besides, how should we store this information to minimize space consumption within acceptable probing overhead? These are the main challenges addressed by our elastic hub index, detailed in Section 5.1.

3.2 Backward Search

Next, we clarify the backward search in *HubPPR*. Effective indexing for the backward search is much more challenging than for forward search, for two reasons. First, unlike random walks, back propagations are *stateful*, i.e., each node v is associated with a residue $r(v,t)$. The effects of each propagation, i.e., modifications to v's reserve value and to the residues of neighboring nodes, depend on the value of $r(v,t)$. In particular, when $r(v,t) < r_{max}$, node v does not perform backward propagation at all. Second, unlike a random walk that has only one result (i.e., its destination), a backward propagation can potentially affect all nodes in the graph.

HubPPR accelerates backward search by pre-computing results for *fractional backward propagations (FBPs)*. An FBP is performed using the backward propagation algorithm described in Section 2.2, with one modification: in the initial step, an FBP can assign an arbitrary residue $\tau \le 1$ to any node $u \in V$. Let $FBP(u,\tau)$ denote the FBP with initial residue τ assigned to node u. For each node $v \in V$, let $r(v,u,\tau)$ (resp. $\pi^{\dashv}(v,u,\tau)$) denote the final residue (resp. reserve) after $FBP(u,\tau)$ terminates. The following lemma describes how pre-computed fractional backward propagation results can be utilized in backward search.

LEMMA 2. *Suppose that, given a node u and an initial residue τ, the results of a fractional backward propagation $FBP(u,\tau)$ consist of final residue $r(v,u,\tau)$ and reserve $\pi^{\dashv}(v,u,\tau)$ for each node $v \in V$. If at any point during a backward search from target node t, u's residue $r(u,t)$ satisfies $r_{max} < r(u,t) \le \tau$, then, recursively propagating u's residue is equivalent to (i) setting residue $r(u,t)$ to $\frac{r(u,t)}{\tau} \cdot r(u,u,\tau)$, and (ii) for each node $v \ne u$, incrementing residue $r(v,t)$ by $\frac{r(u,t)}{\tau} \cdot r(v,u,\tau)$, and reserve $\pi^{\dashv}(v,t)$ by $\frac{r(u,t)}{\tau} \cdot \pi^{\dashv}(v,u,\tau)$.* □

PROOF SKETCH. From the definition of backward search, if an unit information is propagated from u, then eventually $\pi(s,u)$ is propagated to node s. By scaling it with τ, it can be derived that, if τ information is propagated from u, eventually $\tau \cdot \pi(s,u)$ is propagated to s. As a result, we have the following equation.

$$\pi(s,u) = \frac{1}{\tau} \cdot \pi^{\dashv}(s,u,\tau) + \frac{1}{\tau} \cdot \sum_{v \in V} \pi(v,u) \cdot r(v,u,\tau).$$

Then by applying Equation 3 with the current residue and reserve states, and replacing $\pi(s,u)$ with the above equation, we obtain the desired results in Lemma 2. □

Based on Lemma 2, we define the backward oracle as follows.

DEFINITION 3 (BACKWARD ORACLE). *A backward oracle $\mathcal{B}$ is a data structure such that for any input node $u \in V$ and its current residue $r(u,t) > r_{max}$, $\mathcal{B}$ either returns NULL, or* (i) *an initial residue $\tau \geq r(u,t)$, and* (ii) *results of $FBP(u,\tau)$, i.e., $r(v,u,\tau)$ and $\pi^{\dashv}(v,u,\tau)$ for each node $v \in V$ with either $r(v,u,\tau) > 0$ or $\pi^{\dashv}(v,u,\tau) > 0$.* □

Given a backward oracle $\mathcal{B}$, *HubPPR* accelerates backward search as follows. When the search needs to propagate a node u's residue $r(u,t)$, *HubPPR* probes $\mathcal{B}$ with the pair $\langle u, r(u,t)\rangle$. If $\mathcal{B}$ returns NULL, *HubPPR* executes the propagation as usual. Otherwise, *HubPPR* skips the propagation of node u and all subsequent steps, and directly updates each node's residue and reserve according to Lemma 2.

EXAMPLE 3. Consider the backward search in Figure 2. The search starts from t, and probes the backward oracle $\mathcal{B}$ with $\langle t, 1\rangle$. If $\mathcal{B}$ returns NULL, then the search propagates 0.4 and 0.2 information to v_1 and v_2, respectively. Afterwards, $r(v_1,t)$ becomes 0.4. Assume that the next backward search starts from v_1, and it probes $\mathcal{B}$ with $\langle v_1, 0.4\rangle$. Suppose that $\mathcal{B}$ returns 0.5 for τ, $\pi^{\dashv}(v_3,v_1,\tau) = 0.08$, $r(v_3,v_1,\tau) = 0$, $\pi^{\dashv}(v_1,v_1,\tau) = 0.1$, and the residue and reserve for all other nodes are zero. Then, by Lemma 2, we can directly derive that $\pi^{\dashv}(v_3,t) = 0 + \frac{0.4}{0.5}\cdot 0.08 = 0.064$, $r(v_3,t) = 0$, and $\pi^{\dashv}(v_1,t) = \frac{0.4}{0.5}\cdot 0.1 = 0.08$. Afterwards, the search continues backward propagations until all residue are less than r_{max}. □

Challenges in designing $\mathcal{B}$. Similar to the case of $\mathcal{F}$, $\mathcal{B}$ needs to be effective in reducing backward propagation costs, and efficient in terms of both space and time. Compare to $\mathcal{F}$, $\mathcal{B}$ is more complicated since each probe contains not only a node but also a residue value, which drastically increases the space of possible probes. Further, in order for $\mathcal{B}$ to respond to a probe with a non-NULL value, it must contain the corresponding residue and reserve values as in Definition 3, possibly for all nodes in the graph. Hence, it is a challenge to design any non-trivial $\mathcal{B}$ within limited memory space.

Furthermore, the overhead of probing $\mathcal{B}$ and updating residue and reserve values using Lemma 2 should not exceed the cost of performing the corresponding backward propagation. This is not always true for all $\mathcal{B}$ outputs. For instance, if $r(u,t)$ is small and $\mathcal{B}$ returns a much larger τ, the corresponding FBP results may involve considerably more nodes than the actual backward propagation from u. These challenges are addressed later in Section 5.3.

3.3 Complete Algorithm and Analysis

Algorithm 1 demonstrates the pseudo-code of our *HubPPR* algorithm for approximate PPR computation. The algorithm takes as input a forward oracle $\mathcal{F}$ and a backward oracle $\mathcal{B}$, and uses them during forward (lines 9-16) and backward searches (lines 1-8) respectively. Specifically, *HubPPR* utilizes $\mathcal{B}$ to prune backward propagation operations (line 8), and $\mathcal{F}$ to terminate a random walk early (line 16). Note that the oracles do not contain complete information and can return NULL for certain probes, in which case *HubPPR* proceeds as in *BiPPR*. The following theorems establish the correctness and time complexity of *HubPPR* algorithm.

THEOREM 1 (CORRECTNESS OF HUBPPR). *Given a result threshold δ, an approximation ratio ϵ, and a failure probability p_f, with $r_{max} = \sqrt{\frac{m\cdot\varepsilon^2\delta}{n\log(1/p_f)}}$ and $\omega = O\left(\frac{r_{max}}{\varepsilon^2\delta}\log\frac{1}{p_f}\right)$,* HubPPR *is an approximate algorithm for PPR queries.* □

PROOF. Lemma 1 (resp. Lemma 2) shows that the forward (resp. backward) search with the forward (resp. backward) oracle provides identical results as the forward (resp. backward) search in *BiPPR*. Therefore, given the same parameter setting, *HubPPR* and *BiPPR* provide the same approximation guarantee. □

Algorithm 1: *HubPPR*

```
input : $s$, $t$, graph $G$, $\mathcal{F}$ and $\mathcal{B}$
output: $\pi(s,t)$
1  Initialize residue $r(t,t)$ to 1 and $r(v,t)$ to 0 for all node $v \neq t$;
2  Initialize reserve $\pi^{\dashv}(v,t)$ to 0 for all $v$ in $G$;
3  while $\exists u$ satisfying $r(u,t) > r_{max}$ do
4      Prob $\mathcal{B}$ with $\langle u, r(u,t)\rangle$;
5      if $\mathcal{B}$ returns NULL then
6          Perform backward propagation for $u$;
7      else
8          Update the residue and reserve for each node $v$ in $G$
           according to Lemma 2;
9  for $i = 1$ to $\omega$ do
10     Start a new random walk $P$ at node $s$;
11     while $P$ does not terminate do
12         Probe $\mathcal{F}$ with the current node of $P$;
13         if $\mathcal{F}$ returns NULL then
14             Perform one step of random walk on $P$;
15         else
16             Terminate $P$ at the destination node returned by $\mathcal{F}$;
17 Combine backward (lines 1-8) and forward search (lines 9-16) results
   to answer the PPR query with Equation 4;
```

THEOREM 2 (TIME COMPLEXITY OF HUBPPR). *Suppose that each probe of $\mathcal{F}$ takes constant time, and the time complexity of probing $\mathcal{B}$ does not exceed that of performing the corresponding backward propagation. Then,* HubPPR *answers an approximate PPR query in $O\left(\frac{1}{\alpha\varepsilon}\sqrt{\frac{m}{n\delta}}\log\frac{1}{p_f}\right)$ amortized time.* □

PROOF. Note that *HubPPR* improves over *BiPPR* with the use of the forward and backward oracles, neither of which increases its time complexity. As our index could be arbitrary small, in the worst case the amortized time complexity is the same as *BiPPR*. Hence, the time complexity of *HubPPR* is the same as that of *BiPPR*. □

For Theorem 2 to hold, the forward and backward oracles must satisfy the corresponding requirements in the theorem. This is realized in our elastic hub index, clarified in Section 5.

4. TOP-K PERSONALIZED PAGERANK

This section clarifies how *HubPPR* processes top-k PPR queries. Since exact PPR values are computationally expensive to obtain, we focus on deriving approximate top-k answers, in a manner similar to Definition 1. Given a source node s and a target node set T, let t_i^* be the node with the i-th ($k \geq i \geq 1$) largest PPR value from s. We aim to derive a sequence of k nodes $t_1, t_2, \ldots, t_k$, and their estimated PPR values $\hat{\pi}(s,t_1), \hat{\pi}(s,t_2), \ldots, \hat{\pi}(t_k)$ such that

$$|\hat{\pi}(s,t_i) - \pi(s,t_i)| \leq \epsilon/2\cdot\pi(s,t_i), \quad (5)$$

$$|\pi(s,t_i) - \pi(s,t_i^*)| \leq \epsilon\cdot\pi(s,t_i^*), \quad (6)$$

holds with at least $1-p_f$ probability for any $\pi(s,t_i^*) > \delta$.

4.1 Overview

As mentioned in Section 2, a naive approach to answer an approximate top-k PPR query is to (i) perform one approximate PPR query for each node in T, and then (ii) return the k nodes with the largest approximate PPR. However, this incurs significant overheads, due to the large number of approximate PPR queries performed. To address this issue, we propose an iterative approach for top-k queries, such that each iteration (i) eliminates some nodes in T that cannot be top-k results, and (ii) refines the PPR values of the remaining nodes before feeding them to the next iterations. In other words, we pay relatively small processing costs on the nodes that are not top-k results, which helps improve query efficiency.

Specifically, for each node t in the target set T, we maintain a lower bound $LB(t)$ and an upper bound $UB(t)$ of its PPR. Let $\hat{\pi}(s,t) = (LB(t) + UB(t))/2$. We have the following lemma:

LEMMA 3. *If $(1+\epsilon) \cdot LB(t_i) \geq UB(t_i)$, then $\pi(s,t_i)$ and $\hat{\pi}(s,t_i)$ satisfy that:*

$$(1-\epsilon/2)\cdot \pi(s,t_i) \leq \hat{\pi}(s,t_i) \leq (1+\epsilon/2)\cdot\pi(s,t_i), \quad (7)$$

$$(1-\epsilon)\cdot\pi(s,t_i^*) \leq \pi(s,t_i) \leq (1+\epsilon)\cdot\pi(s,t_i^*), \quad (8)$$

where t_i is the node with the i-th largest PPR lower bound, and t_i^* is the node with the i-th largest exact PPR value. □

In other words, if $(1+\epsilon)\cdot LB(t_i) \geq UB(t_i)$ holds for every $i \in [1,k]$, then we can return the k nodes in T with the largest PPR lower bounds, and their estimation $\hat{\pi}(s,t_i)$ as the answer of the approximate top-k PPR query.

4.2 Algorithm

Recall that our top-k method runs in an iterative manner. In a nutshell, the i-th iteration of this method consists of three phases:

- *Forward phase.* This phase performs forward searches from the source node s using 2^i random walks.
- *Backward phase.* This phase performs backward searches from selected target nodes in T, such that r_{max} value for each selected node is half of its value in the $(i-1)$-th iteration. (The initial r_{max} value for each node is set to 1.) In other words, it increases the accuracy of the backward searches for the selected nodes.
- *Bound estimation.* This phase updates the PPR lower and upper bounds of all target nodes, and decides whether the algorithm should terminate, based on Lemma 3.

Algorithm 2 shows the pseudo-code of our approximate top-k PPR query algorithm. Initially, the number of random walks is set to 1, and r_{max} for each target t in T is set to the maximum value 1 (Line 1). Afterwards, the algorithm initializes the costs f_c (resp. b_c) for the forward (resp. backward) phase to zero (Line 2), where the forward (resp. backward) cost is defined as the total number of jumps during the random walks (resp. the total number of residual / reserve updates in the backward phase). Then, the two costs are updated in each iteration, and the algorithm alternates between the forward and backward phases in a manner that balances their computation costs, which is important to preserve the time complexity as will be shown in Theorem 3.

A list C is maintained to include the nodes t' that are candidates for the top-k answers, and is initialized to include all nodes (Line 3). In each iteration, it eliminates the nodes that will not be the top-k answers, i.e., $UB(t') < LB(t_k)$ (Line 18). This strategy iteratively eliminates the nodes that are not top-k answers, and saves the query time. Lines 6-9 of Algorithm 2 demonstrate the backward phase in an iteration. It repeatedly selects the node t from C such that the gap ratio between its lower bound and upper bound is minimum (Line 6), i.e., the node that has the most loose bound in C. Then, it halves the r_{max} value. Let $r_{max}(t,i)$ denote the r_{max} value of node t in the i-th iteration. It then continues the backward propagation from t until the residual for all nodes are smaller than $r_{max}(t,i)$, and updates the backward cost b_c. The backward phase continues until the backward cost b_c is no smaller than the forward cost f_c. When the backward phase finishes, the forward phase then generates 2^i random walks and updates the forward cost f_c (Lines 11-12). Finally, it derives the lower and upper bounds for each node in T (Lines 13-17). The details of how to derive the bounds are more involved, and we elaborate on it in Section 4.3.

Algorithm 2: Approximate top-k query algorithm (s, T)

```
Input: source $s$, target set $T$
Output: $k$ nodes with the highest $k$ PPR score in $T$
1  Let $\omega \leftarrow 1$, $r_{max}(t,0) \leftarrow 1$ for all $t \in T$;
2  forward cost $f_c \leftarrow 0$, backward cost $b_c \leftarrow 0$;
3  initialize the candidate list $C$ to include all nodes in $T$;
4  for $i = 1$ to $\infty$ do
5      while $f_c > b_c$ do
6          Select a candidate node $t$ from $C$ such that $LB(t)/UB(t)$ is
           the minimum among all nodes in $T$;
7          $r_{max}(t,i) \leftarrow \max(r_{max}(t,i-1)/2, r_{max})$;
8          Continue the backward propagation until the reserve of each
           node $t'$ is smaller than $r_{max}(t',i)$;
9          $b_c \leftarrow b_c + b'$, where $b'$ denotes the backward cost;
10     Generate $\omega_i = 2^i$ random walk from $s$ ;
11     Let $f_\omega$ be the cost for the $\omega_i$ random walks, $f_c \leftarrow f_c + f_\omega$;
12     Compute $LB(t), UB(t)$ for each $t$ in $C$ using Lemma 5;
13     Let $t_j$ be the node with the $j$-th largest LB in $C$;
14     if $LB(t_j)\cdot(1+\varepsilon) \geq UB(t_j)$ for all $j \in [1,k]$ then
15         return the $k$ nodes in decreasing order of $LB(t)$;
16     if $\forall t \in T$, $\omega_i \geq \frac{2r_{max}}{\epsilon^2\cdot\delta}\cdot\log(2\cdot k/p_f) \wedge r_{max}(t) \leq r_{max}$
       then
17         return $k$ random nodes;
18     Eliminate the nodes $t'$ from $C$ such that $UB(t') < LB(t_k)$;
```

The above description does not take into account the forward oracle $\mathcal{F}$ and backward oracle $\mathcal{B}$. When $\mathcal{F}$ and $\mathcal{B}$ are present, the algorithm utilizes them similarly as in Algorithm 1. In particular, each forward search exploits $\mathcal{F}$ to accelerate random walks; each backward search uses pre-computed fractional backward propagations to reduce search costs. We omit the details for brevity.

4.3 Bound Estimation

In what follows, we clarify how we derive the PPR lower bound $LB(t)$ and upper bound $UB(t)$ for each node $t \in T$. Our derivation is based on *martingales* [28]:

DEFINITION 4 (MARTINGALE). *A sequence of random variables $Y_1, Y_2, Y_3, \cdots$ is a martingale iff $\mathbb{E}[|Y_j|] \leq +\infty$ and $\mathbb{E}[Y_j | Y_1, Y_2, \cdots, Y_{j-1}] = \mathbb{E}[Y_{j-1}]$.* □

Given the backward propagation results $r(v,t)$ for $v \in V$ with target node t, let $Z_1, Z_2, \cdots, Z_j, \cdots$ be the ending nodes of the random walks generated in Algorithm 2, we define a sequnce of random variables $X_1, X_2, \cdots, X_j, \cdots$ such that $X_j = r(Z_j,t) - \sum_{v\in V} r(v,t)\cdot\pi(s,v)$. Then it is easy to verify that $\mathbb{E}[X_j] = 0$. Define $M_j = X_1 + X_2 \cdots + X_j$. We have $\mathbb{E}[X_j] = \mathbb{E}[M_j] = 0$. As the new sampled random walks is independent from all previous sampled random walks (although the decision of whether generating the j-th random walk depends on $X_1, X_2, \cdots X_{j-1}$), we have $\mathbb{E}[M_j \mid M_1, M_2, \cdots, M_{j-1}] = \mathbb{E}[M_{j-1}]$. As a consequence, $M_1, M_2, \cdots, M_j, \cdots$, is a sequence of martingale. The following lemma shows an important property of martingales:

LEMMA 4 ([10]). *Let $Y_1, Y_2, Y_3 \cdots$ be a martingale, such that for any i ($1 \leq i \leq \omega$), we have $|Y_i - Y_{i-1}| \leq a_i + \Delta$, and $\mathrm{Var}[Y_i \mid Y_1, Y_2, \cdots, Y_{i-1}] \leq \sigma_i^2$. Then,*

$$\Pr[|Y_\omega - \mathbb{E}[Y_\omega]| \geq \lambda] \leq \exp\left(-\frac{\lambda^2}{2(\sum_{j=1}^{\omega}(\sigma_j^2 + a_j^2) + \Delta\cdot\lambda/3)}\right).$$

Next, we demonstrate how to make a connection from our problem to Lemma 4. Let Ω_i be the total number of random walks sampled in the first i iterations. Then, our goal is to derive the lower and upper bounds for each node in T in the i-th iteration. Denote the r_{max} value for target t in the i-th iteration as $r_{max}(t,i)$. Then, in the i-th iteration, we set $\Delta = r_{max}(t,i)$. We further set $a_j = 0$.

With this setting, it can be guaranteed that $|M_j - M_{j-1}| \le a_j + \Delta$. Meanwhile, for M_j, we also have the following equation:

$$\mathrm{Var}[M_j \mid M_1, M_2, \cdots, M_{j-1}] \le r_{max}(t,i)^2/4.$$

Then, for each j, we set $\sigma_j^2 = r_{max}(t,i)^2/4$. Let $b = \sum_{j=1}^{\omega}(\sigma_j^2 + a_j^2)$ and $M_{\Omega_i} = \sum_{j=1}^{\Omega_i} r(Z_i, t)$. Applying Lemma 4, we have Lemma 5.

LEMMA 5. *Let $p_f^* = \frac{p_f}{2|T|\cdot\log(n^2\cdot\alpha\cdot|T|)}$, and*

$$\lambda = \sqrt{\left(\tfrac{2\cdot\Delta}{3}\ln p_f^*\right)^2 - 2b\cdot\ln p_f^*} - \tfrac{2\cdot\Delta}{3}\ln p_f^*.$$

Then, with $1 - p_f^*$ probability, in the i-th iteration, we have

$$\max\{0, M_{\Omega_i} - \lambda\}/\Omega_i \le \pi(s,t) - \pi^{\dashv}(s,t) \le \min\{1, M_{\Omega_i} + \lambda\}/\Omega_i.$$

Based on Lemma 5, we set $LB(t) = \pi^{\dashv}(s,t) + \max(0, M_{\Omega_i} - \lambda)/\Omega_i$ and $UB(t) = \pi^{\dashv}(s,t) + \min(1, M_{\Omega_i} + \lambda)/\Omega_i$, which are correct bounds for $\pi(s,t)$ with at least $1 - p_f^*$ probability.

4.4 Approximation Guarantee

As shown in Algorithm 2, we calculate $LB(t)$ and $UB(t)$ multiple times, and we need to guarantee that all the calculated bounds are correct so as to provide the approximate answer. The following corollary demonstrates the probability that all the LB (resp. UB) bounds in Algorithm 2 are correct.

COROLLARY 1. *When Algorithm 2 terminates, the probability that PPR bounds for all target nodes are correct is at least $1-p_f/2$.*

Given all the correct bounds, to return the approximate top-k answers, it requires that Algorithm 2 terminates at Line 15 instead of Line 17. We have the following lemma to show the probability that Algorithm 2 terminates at Line 15.

LEMMA 6. *Algorithm 2 terminates at Line 15 with at least $1 - p_f/2$ probability.*

Combining Lemma 3, Lemma 6, and Corollary 1, we have the following theorem for the approximation guarantee and the average time complexity for our top-k PPR query algorithm.

THEOREM 3. *Algorithm 2 returns k nodes $t_1, t_2, \cdots, t_k$, such that Equations 5 and 6 hold for all t_i^* ($i \in [1,k]$) whose PPR $\pi(s, t_i^*) > \delta$ with at least $1 - p_f$ probability, and has an average running time of $O\left(\frac{1}{\alpha\cdot\varepsilon}\sqrt{\frac{m\cdot|T|}{n\delta}}\log\frac{k}{p_f}\right)$, when the nodes in T are chosen uniformly at random.* □

PROOF SKETCH. Based on Lemma 6 and Corollary 1, it can be verified that the returned nodes satisfy the approximation with probability at least $1 - p_f$. Meanwhile, the amortized time complexity of the algorithm can be bounded based on (i) the number of random walks shown in Lemma 6 and (ii) the amortized cost $O\left(\frac{m\cdot|T|}{n\cdot\delta\cdot\alpha}\right)$ for performing backward search from $|T|$ nodes. □

5. ELASTIC HUB INDEX

The *elastic hub index (EHI)* contains both a forward oracle and a backward oracle, whose functionalities and requirements are explained in Section 3. The EHI resides in main memory, and it can utilize any amount of available memory space; meanwhile, the size of the EHI can be dynamically adjusted by shrinking or expanding the oracles. In particular, both forward and backward oracles contain information on hub nodes, and their size can be controlled by adjusting the number of hubs. Meanwhile, information on different hubs can be computed in parallel. In the following, Sections 5.1 and 5.2 present the forward oracle and the selections of hubs therein. Sections 5.3 and 5.4 deal with the backward oracle.

5.1 Forward Oracle

The forward oracle $\mathcal{F}$ contains pre-computed random walk destinations for a selected set of nodes H_f, which we call *forward hubs*. Specifically, for each forward hub $h \in H_f$, $\mathcal{F}$ contains $\mu = \omega$ destinations of random walks starting from node v, denoted as $\mathcal{F}(h)$. Note that ω is the maximum number of destinations to store at each hub, which ensures that a random walk reaching the hub can immediately terminate. As will be shown in Section 5.2, forward hubs added earlier to the index are expected to have higher pruning power than later ones. Therefore, intuitively earlier hubs should be allocated more destinations than later ones, and the current design of always allocating the maximum number of destinations is based on this observation.

When probed with a non-hub node $v \notin H_f$, $\mathcal{F}$ always returns NULL; otherwise, i.e., when probed with a hub node $h \in H_f$, $\mathcal{F}$ returns a different destination in $\mathcal{F}(h)$ for each such probe. Note that $\mathcal{F}$ can respond to probes with h at most μ times; after that, $\mathcal{F}$ returns NULL for any further probes with h.

Next we focus on the data structure for $\mathcal{F}(h)$. As mentioned in Section 3.1, storing $\mathcal{F}(h)$ as a list wastes space due to duplicates. One may wonder whether we can compress this list into a multiset, in which each entry is a pair $\langle v, c_v\rangle$ consisting of a destination v and a counter c_v recording the number of times that v appears in $\mathcal{F}(h)$. For example, if the destinations pre-computed for h are v_1, v_1, v_2, v_1. Then with multiset, it can be stored as $S = \{\langle v_1, 3\rangle, \langle v_2, 1\rangle\}$. The problem with this approach, however, is that the multiset loses the information on the *order* of the destinations. For example, if we directly return the first two nodes in S as the destinations for the two probes, the random walks are then not independent, violating the requirement for Chernoff bound, and can not provide the approximation guarantee.

To address the above problem, first we observe that $\mathcal{F}$ can return destinations in $\mathcal{F}(h)$ in any random order, as follows.

OBSERVATION 1. *Given a forward hub $h \in H_f$ and the corresponding random walk destinations $\mathcal{F}(h)$, for each probe with h, $\mathcal{F}$ can select and return a random node $v \in \mathcal{F}(h)$ among those that have not been used to answer previous probes, without affecting the correctness of* HubPPR.

One way to report all μ nodes represented by a multiset in a random order is to *(i)* draw a random node v from the multiset for each probe with probability proportional to its associated counter c_v, and *(ii)* decrease c_v for v by 1 if $c_v > 1$, or delete $\langle v, c_v\rangle$ from the multiset if $c_v = 1$. The problem with this method, however, is that deleting an element from a multiset takes $O(\log\mu)$ time, which violates the constant-time probe requirement. Hence, we need a more sophisticated data structure than a multiset. We next make the following observation.

OBSERVATION 2. *When $\mathcal{F}$ responds to a probe with a random walk destination v, it can do so* asynchronously*, i.e., it first notifies* HubPPR *that its response is not NULL, and then returns v after all random walks terminate in the forward search.*

According to the above observation, for each forward hub $h \in H_f$, $\mathcal{F}$ can collect the number of probes k_h with h, and return a batch of k_h random destinations in $\mathcal{F}(h)$ to *HubPPR* towards the end of the forward search. To simplify our notations, in the following we focus on a particular hub h and omit the subscript h, e.g., we assume that h is probed k times.

The main idea of the proposed data structure for $\mathcal{F}$ to divide $\mathcal{F}(h_f)$ into several disjoint multi-sets $\mathcal{S} = \{F_1, F_2, \cdots, F_i\}$ such that $F_1 \cup F_2 \cup \cdots \cup F_i = \mathcal{F}(h_f)$. When selecting k nodes from

$\mathcal{F}(h_f)$, we find some different multi-set $F'_1, F'_2, \cdots F'_j$ from $\mathcal{S}$ such that for their merged multi-set $F' = \{F'_1 \cup F'_2 \cup \cdots F'_j\}$, it satisfies that $|F'| = k$. Note that as long as we are selecting k different random walks, they are guaranteed to be independent, meaning that they do not affect the correctness of *HubPPR*.

Next we clarify how we divide $\mathcal{F}(h_f)$. The solution should guarantee that for an arbitrary k, we can always find some different multi-sets in $\mathcal{S}$ such that the size of their merged result is k. To achieve this, we propose to divide $\mathcal{F}(h_f)$ into $u = 1 + \lfloor \log_2 \ell \rfloor$ multi-sets $\mathcal{S} = \{F_1, F_2, \cdots, F_u\}$, where the i-th multi-set F_i contains 2^{i-1} nodes for $1 \leq i \leq u$, and the last multi-set F_u contains the remaining $\ell + 1 - 2^{u-1}$ nodes. Regarding $\mathcal{S}' = \{F_1, F_2, \cdots, F_{u-1}\}$, we have the following theorem.

THEOREM 4. *For any number $k \leq 2^{u-1} - 1$, we can find a set $\mathcal{C}' \subseteq \mathcal{S}'$, such that the $\sum_{F_i \in \mathcal{C}'} |F_i| = k$.*

PROOF. Let the binary code of k be X. Clearly, the number of digits in X is no more than $u - 1$. If the j-th ($1 \leq j \leq u - 1$) digit is 1, then F_j is selected. Note that if the j-th digit is 1, it contributes 2^j to the final sum, i.e., k. Meanwhile, the size of F_j is exactly 2^j. Let $\mathcal{C}'$ be the set of multi-sets that we selected. Then it is guaranteed that $\sum_{F_i \in \mathcal{C}'} |F_i| = k$, which finishes the proof. □

With the above theorem, for any $k \leq 2^{u-1} - 1$, we can easily find some multi-sets in $\mathcal{S}'$ such that the size of their merged result is k. For a $k > 2^{u-1} - 1$, we proceed in the following way. We first select F_u; then we select $k - |F_u|$ nodes using Theorem 4. The reason is that $k - |F_u|$ will always be no larger than $2^{u-1} - 1$, which makes it satisfy the condition of Theorem 4. When the sets are selected, we merge them and return the merged set. It is guaranteed that F includes k sampled node from $\mathcal{F}(h_f)$, and they are all independent.

To analyze the cost, to find the multi-sets, it requires $O(\log k)$ cost. To return the set of k ending nodes, it requires to merge the result from several different multi-sets, which can be bounded by $O(k)$. As a consequence, the total cost for selecting k random nodes is $O(k)$.

It remains to clarify how to store the disjoint multi-sets in $\mathcal{S}$. As we record multi-set with different sizes, the tuple based solution may not be ideal. For example, when we store a set which only includes two different nodes, then we require $4 \times 4 = 16$ bytes to store it with the tuple based solution, assuming that it requires four bytes to record the nodes / counts. In contrast, if we use an array to store the nodes, it only consumes $4 \times 2 = 8$ bytes to store the nodes. As a result, we adopt a hybrid scheme to store each multi-set: when the space consumption of an array-based solution is less than the tuple-based solution, then we use the array-based storage; otherwise, we adopt the tuple-based solution. The following example shows how the compression scheme works.

EXAMPLE 4. *Given $h_1 \in H_f$, suppose that 12 random walks are generated from node h_1, and the ending nodes are $\{v_1, v_1, v_1, v_1, v_1, v_1, v_1, v_1, v_3, v_3, v_3, v_3\}$. Then the nodes are first divided into four parts, i.e., $F_1 = \{v_1\}, F_2 = \{v_1, v_1\}, F_3 = \{v_1, v_1, v_1, v_1\}, F_4 = \{v_1, v_3, v_3, v_3, v_3\}$. For each part, the nodes are stored as a multi-set with the hybrid scheme, i.e., $\{v_1\}, \{(v_1, 2)\}, \{(v_1, 4)\}, \{(v_1, 1), (v_3, 4)\}$. Observe that with our hybrid scheme, the space consumption is 9×4 bytes, while the list based solution requires at least 12×4 bytes.*

Assume that we have 9 out of ω random walks that end at h_1 and let F be the set to contain the ending nodes of 9 random walks. We first derive the binary code of 9, which is 1001. Here, we can calculate that $u = 1 + \lfloor \log_2 \ell \rfloor = 4$ and $9 > 2^{u-1} - 1 = 7$. So we first add the nodes in F_4 into F and then select $9 - |F_4| = 4$ nodes. As the binary code of 4 is 100, we add the nodes in F_3 into F, ending up with $F = \{(v_1, 5), (v_3, 4)\}$. □

Algorithm 3: Forward Hub Selection

```
Input: Graph $G$, probability$\alpha$, the number of forward hubs $\kappa$
Output: The set of forward hub $H_f$
1  $H_f \leftarrow \emptyset$;
2  Generate $\omega' = \frac{3\log(2/p_f)}{\delta\epsilon^2}$ random walks.
3  For each node $v$, compute the total number of saved hops $B(v)$ by $v$.
4  do
5      Select a node $v$ with highest $B(v)$ score and add it into $H_f$;
6      for each random walk $W$ that visits $v$ do
7          Let $l$ denote the number of visited nodes in $W$;
8          Let $i$ denote the position that $v$ first appears in $W$;
9          Let $u_c$ denote the vertex that appears firstly in the $c$-th
           position in $W$;
10         if $c < i$ then
11             $B(u_c) \leftarrow B(u_c) - (l - i)$;
12         else
13             $B(u_c) \leftarrow B(u_c) - (l - c)$;
14         Update $W$ by removing the nodes from $i$-th position to the
           $l$-th position ;
15 while $|\mathcal{F}| \leq L_f$;
```

5.2 Selection of Forward Hubs

Next, we present how to select the forward hubs. Let L_f be the memory size allocated to the forward oracle. Our goal is to to maximize the effectiveness of the forward oracle $\mathcal{F}$ within L_f space. In particular, we model this goal as an optimization problem that aims to maximize the expected number of eliminated hops during the random walks during forward search, under space constraint L_f. This optimization problem, however, is difficult to solve exactly, since the number of possible random walks can be exponential to graph size. Therefore, we use a sampling-based greedy algorithm to select the forward hubs, as shown in Algorithm 3.

Initially, we sample ω random walks and record the total number of hops $B(v)$ that can be saved by node v if v is selected as a forward hub (Lines 2-3). In particular, for each random walk W, let c be the first position that v appears in the random walk $W = (v_1, v_2, \cdots, v_i, \cdots, v_l)$, the number of saved hops on W is then $l - c$. As a result, we increment $B(v)$ by $l - c$ for random walk W.

After $B(v)$ is initiated for each node v, Algorithm 3 iteratively selects the node that has the maximum $B(v)$ as a forward hub (Line 5). When a node is selected in an iteration, the number of saved hops for other nodes are updated (Lines 7-14). Specifically, for each random walk W that visits v, it updates $B(v)$ for all the other nodes. Let l be the number of nodes visited by W, and i be the position that v first appears in W. For each node u_c that first appears at position $c \in W$, if c is smaller than i, then when random walks stops and v, it will stop, and the number of saved hops for u_c is reduced by $l - k$ (Algorithm 3 Lines 10-11). Otherwise, u_c is a successor of v, and the random walk will not visit this node, and the number of saved hops for random walk W is $l - c$ before v is selected as a forward hub, so the number of saved hops for u_c is reduced by $l - c$ (Algorithm 5 Lines 12-13). Afterwards, the random walk W is truncated by removing the subsequent nodes from the i-th position, i.e., the position that v first appears, to the ending nodes of W. Finally, if the index size does not exceed L_f, it continues the next iteration and repeats until no more forward hubs can be added.

EXAMPLE 5. *Consider Figure 3. Assume that 4 random*

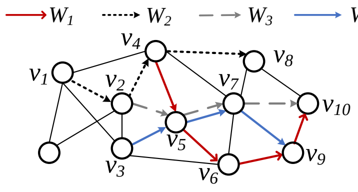


Figure 3: Example of forward hub selection

walks are sampled by Algorithm 3. Then the score for $B(v_1), \cdots, B(v_{10})$ *are initialized to:* $B(v_1) = 3, B(v_2) = 5, B(v_3) = 3, B(v_4) = 4, B(v_5) = 7, B(v_6) = 2, B(v_7) = 2, B(v_8) = 0, B(v_9) = 1, B(v_{10}) = 0$. *As* v_5 *has the highest score* $B(v_5)$*, it is selected as a forward hub. Meanwhile, the benefit score for other nodes are updated. Specifically, the score for* v_6, v_7, v_9 *are updated to zero since the random walks that visit these nodes will stop at* v_5 *and not reach these nodes. Meanwhile, for* v_2*,* $B(v_2)$ *is updated to 3 since* W_3 *goes through* v_2 *and* v_5*, and if* v_2 *is selected as a hub, it will only save 1 hop for* W_3*. Similarly,* $B(v_3)$ *and* $B(v_4)$ *are updated to 1 and 2, respectively. In addition,* $B(v_1)$ *remains untouched since the random walk that goes across* v_1 *does not go through* v_5*. Finally, Algorithm 3 checks whether the size of* $\mathcal{F}$ *exceeds the threshold* L_f*, and continues if more forward hubs can be added.* □

5.3 Backward Oracle

As described in Section 3.2, the backward oracle $\mathcal{B}$ answers probes with both a node u and a residue $r(u,t)$, and it returns either NULL, or an initial residue τ and the results of $FBP(u,\tau)$. Similar to the case of the forward oracle, we select a set of *backward hubs* H_b, and for each such hub $h \in H_b$ we store the results of multiple FBPs originating from h with different initial residue τ which we call *snapshots*.

We first focus on how to determine the snapshots for a given backward hub h. Since we need to materialize results of an FBP for each snapshot, the number of stored snapshots should be minimized to save space. On the other hand, having too few snapshots may violate the probe efficiency requirement. As described in Section 3.2, if the returned τ is much larger than h's residue $r(h,t)$, using $\mathcal{B}$ might lead to a high cost since it involves numerous nodes. To these issues, the propose index construction algorithm (i) builds multiple snapshots for each $h \in H_b$; and (ii) guarantees that the index size is at most twice of that of a single snapshot.

Algorithm 4 shows algorithm for selecting different initial residual values for each $h \in H_b$ (Line 1). In particular, we start with initial residual $\tau = 1$ and perform backward propagation from h (Line 2). We store the snapshot into $\mathcal{B}(h)$ and record the size of the snapshot. Afterwards, we proceed an FBP from h with initial residual 0.5, and check if the size of the snapshot is larger than half of the snapshot $S(h,1)$'s size. If so, we discard the former snapshot. Otherwise, we add it to $\mathcal{B}(h)$ and record its size (Lines 4-10). Afterwards, we repeatedly set the initial residue τ to half of that in previous iteration (Line 11), until the initial residue falls below r_{max} (Line 3).

Because the size of a stored snapshot for h is always no more than half the size of the previous stored snapshot, the total size of all stored snapshots is no larger than twice of the size of the first snapshot, i.e., $S(h,1)$, which satisfies the space consumption bound. Finally, we establish the time efficiency of the proposed backward oracle through the following lemma:

LEMMA 7 (BACKWARD ORACLE COMPLEXITY). *Given a backward oracle* $\mathcal{B}$ *returned by Algorithm 4, it is guaranteed that the amortized cost of the backward search using* $\mathcal{B}$ *is* $O(\frac{m}{n\cdot\alpha\cdot r_{max}})$.

Algorithm 4: Backward Oracle Construction

```
   Input: $H_b$ the set of backward hubs
   Output: Backward Oracle $\mathcal{B}$
1  for each node $h$ in $H_b$ do
2      Let $\tau \leftarrow 1$
3      while $\tau > r_{max}$ do
4          Perform $FBP(h,\tau)$;
5          Let $S(h,\tau)$ denote the snapshot of $FBP(h,\tau)$;
6          Let $S(h,\tau')$ denote the last added snapshot in $\mathcal{B}(h)$;
7          if $|S(h,\tau)| < |S(h_b,\tau')|/2$ then
8              Add $S(h,\tau)$ into $\mathcal{B}(h_b)$;
9          else
10             Ignore this snapshot;
11         Let $\tau \leftarrow \tau/2$;
12 return $\mathcal{B}$
```

5.4 Selection of Backward Hubs

Next we clarify the selection of backward hubs. There are two major considerations for deciding whether to add a node h to $\mathcal{B}$: (i) the space consumption incurred by h's snapshots; and (ii) benefit in terms of query time reduction caused by h as a backward hub. Given a randomly selected target node t, let X be a random variable to denote the number of saved backward propagation operations with h's snapshot, and Y be a random variable to denote the size of the chosen h's snapshot. Then the benefit can be calculated as $\beta\cdot\mathbb{E}[X]-\gamma\cdot\mathbb{E}[Y]$, where β and γ are the cost for a single backward propagation operation and for updating the residue and reserve on one node, respectively.

We restrict the size of $\mathcal{B}$ to be no more than a given value L_b. Suppose the benefit can be efficiently calculated, then we can model backward hub selection as a knapsack problem, and solve it with a greedy algorithm. Specifically, each time, we select the node v with maximum benefit to cost ratio: $\frac{\beta\cdot\mathbb{E}[X_v]-\gamma\cdot\mathbb{E}[Y_v]}{c\cdot\mathbb{E}[Y_v]}$, where c is the number of snapshots a backward hub holds. Meanwhile, the number of snapshots for important nodes, i.e., those with high pruning power, are roughly similar ($\approx \lfloor\log_2\frac{1}{r_{max}}\rfloor$). Hence, the key is then how to calculate or approximate $\mathbb{E}[X_v]/\mathbb{E}[Y_v]$ efficiently. This is rather difficult, however, since for different residue values for h, the selected snapshot for updating will be different, and the number of saved backward propagation operations will be different. This motivates us to devise a heuristic solution to estimate this score.

Our main observations are (i) that the more message that has been propagated from v, the higher the probability it is that could help prune backward push operations, and (ii) the larger the average snapshot size $\bar{\mathcal{B}}(v)$ is for a node v, the larger the number of backward push operations can be saved. Our heuristic then use $\mathbb{E}[M_v]\cdot\mathbb{E}[Y_v]$ to indicate the expected pruning efficiency $\mathbb{E}[X_v]$ for node v. Then, $\mathbb{E}[X_v]/\mathbb{E}[Y_v]$ can be estimated as $\frac{\mathbb{E}[M_v]\cdot\mathbb{E}[Y_v]}{\mathbb{E}[Y_v]} = \mathbb{E}[M_v]$, which can be efficiently approximated with Monte-Carlo methods. Algorithm 5 shows the pseudo-code for how we select the backward hubs. We first initialize the total propagated message $l(v)$ for each node v to zero (Line 1). Next, we randomly select ω ending nodes, proceed backward propagation from these nodes, and record the size $l'(v)$ of the propagated message from each node v for the ω backward propagations (Lines 3-4). Afterwards, we update $l(v)$ for each node by adding $l'(v)$ into it (Line 5). Subsequently, we repeatedly select the node v with the highest $l(v)$ score , and generate the snapshots for v (Lines 8-11). Finally the algorithm stops when the size of the Backward Oracle exceeds the pre-defined threshold L_b (Line 12).

Algorithm 5: Backward Hub Selection

```
   Input: Graph $G$, probability $\alpha$
   Output: the set $H_b$ of backward hubs
1  $l(v) \leftarrow 0$ for all $v \in V$;
2  Randomly select $\omega$ nodes as the target set $T$;
3  Proceed a backward search with all initial reserve as zero, and initial
   residues $r(v, T) = 1$ if $v \in T$ and $r(v, T) = 0$ otherwise;
4  Record the propagated message $l(v)$ from each node $v$;
5  do
6      Select the node $v$ with highest $l(v)$ score;
7      Invoke Algorithm 4 Lines 2-11 to generate snapshots for node $v$;
8      Add $v$ into $H_b$;
9      Create an entry $\mathcal{B}(v)$ and add the generated snapshots into $\mathcal{B}(v)$;
10 while $|\mathcal{B}| \leq L_b$;
```

5.5 Elastic Adjustment

Next, we explain how EHI can be dynamically adjusted.

First, note that the elastic hub index (EHI) in *HubPPR* can be constructed incrementally upon an existing one. For example, when we have an EHI with 5x space ratio, and we want to construct an EHI with 10x space ratio, it does not need to construct the new index from scratch. To explain, we can first calculate a total order for the nodes to indicate its importance in forward (resp. backward) search using the sampling based approach in forward (resp. backward) hub selection algorithm. Afterwards, the index can be constructed incrementally based on these two total orders. Given a 5x space *HubPPR* index constructed based the two total orders, assuming that the last forward (resp. backward) hub in the index has a total order i (resp. j), then we can reuse the 5x space *HubPPR* index, and start constructing the forward oracle (resp. backward oracle) from the node whose total order is $i+1$ (resp. $j+1$), and increase the index size until the newly added index reaches 5x space, adding up to 10x space in total.

Besides, to shrink the index, we could simply stop loading the index when it reaches the specified memory capacity. For example, given an EHI with 5x graph size, and one wants to use only 4x-graph-size memory, then, we can load the index hub by hub, and stop loading the index when it reaches the space threshold.

6. OTHER RELATED WORK

PageRank and Personalized PageRank are first introduced by Page et al. [25]. The former measures the global importance of a node in the graph, and the latter measures the importance of a node with respect to another node. Both problems have been extensively studied. We focus on the Personalized PageRank, and refer readers to [7, 23] for detailed surveys on PageRank.

In [25], Page et al. propose the *power iterations* approach to calculate the exact PPR vector with respect to a source node s using Equation 1, where the PPR vector includes the PPR values for all $v \in V$ with respect to s. As explained in Section 2.2, this methods involves matrix operations on the adjacency matrix, which incurs high space and time costs for large graphs. Subsequently, a branch of research work focuses on developing algorithms to efficiently compute PPR vectors [8, 9, 11, 13, 18, 22, 29]. Jeh et al. [18] propose the backward search solution as discussed in Section 2.2, which is further optimized in [2, 11]. Berkhin [8], Chakrabarti [9], and Zhu et al. [29] propose to (i) pre-compte the PPR vectors for some selected hub nodes, and then (ii) use the pre-computed results to answer PPR queries. However, Berkhin's method is limited to the case when the source is a distribution where all non-hub nodes have zero probabilities; Chakrabarti's and Zhu et al.'s techniques are based on variants of power iterations [25] for PPR computation, and, thus, inherit its inefficiency for large graphs. Fujiwara et al. [13] propose to pre-compute a QR decomposition of the adjacency matrix A (see Section 2), and then utilize the results to accelerate PPR queries; nonetheless, the method incurs prohibitive preprocessing costs on million-node graphs. Later, Maehara et al. [22] present a method that (i) decomposes the input graph into a core part and several tree-like structures, and then (ii) exploits the decomposition to speed up the computation of exact PPR vectors. Shin et al. [27] propose *BEAR* to reorder the adjacency matrix of the input graph G to obtain several easy-to-invert sparse sub-matrices. The sub-matrices are then stored as the index, and used to improve PPR query processing. This approach incurs prohibitive space consumption, and they further propose *BEAR-Approx* to reduce the index size by dropping values that are less than a dropout threshold γ in the sub-matrices and setting them to zero. Nevertheless, as will be shown in our experiment, *BEAR-Approx* still incurs prohibitive preprocessing costs, and is not scalable to large graphs.

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

Name	n	m	Type	Linking Site
DBLP	613.6K	2.0M	undirected	www.dblp.com
Web-St	281.9K	2.3M	directed	www.stanford.edu
Pokec	1.6M	30.6M	directed	pokec.azet.sk
LJ	4.8M	69.0M	directed	www.livejournal.com
Orkut	3.1M	117.2M	undirected	www.orkut.com
Twitter	41.7M	1.5B	directed	twitter.com
UkWeb	105.9M	3.7B	directed	—

In addition, the random walk based definition of PPR inspires a line of research work [6, 11, 20, 21, 26] that utilizes the Monte-Carlo approach to derive approximate PPR results. In particular, Bahmani et al. [5] and Sarma et al. [26] investigate the acceleration of the Monte-Carlo approach in distributed environments. Fogaras et al. [11] presents a technique that pre-computes *compressed* random walks for PPR query processing, but the large space consumption of the technique renders it applicable only on small graphs. Lofgren et al. propose *FastPPR* [21], which significantly outperforms the Monte-Carlo method in terms of query time. *FastPPR*, in turn, is subsumed by *BiPPR* [20] in terms of query efficiency.

There also exists a line of research work [4, 6, 12, 13, 15, 20] that investigates top-k PPR queries. Nevertheless, almost all existing methods require that the target set $T = V$, i.e., the set of all nodes in the input graph. The only algorithm that supports arbitrary T is *BiPPR*. In the next section, we show that *HubPPR* significantly outperforms *BiPPR* through extensive experimental results.

7. EXPERIMENTS

In this section, we experimentally evaluate the *HubPPR* framework for both PPR and top-k PPR queries. Section 7.1 explains the experimental settings. Section 7.2 evaluates HubPPR for PPR queries against the state-of-the-art methods. Section 7.3 evaluates HubPPR for top-k PPR processing. Section 7.4 provides insights for choosing the appropriate index size for *HubPPR*.

7.1 Experimental Settings

All the experiments are tested on a Linux machine with an Intel Xeon 2.4GHz CPU and 256GB RAM. We repeat each experiment 5 times and report the average results.

Datasets. We use 7 real datasets in our evaluations, containing all 6 datasets used in [20, 21]. Among them, DBLP, Pokec, LiveJournal (abbreviated as LJ), Orkut, and Twitter are social networks, whereas UKWeb is a web graph. Besides these, we use one more web graph: Web-Stanford (denoted as Web-St), adopted from SNAP [19]. Table 2 summarizes the statistics of the 7 datasets.

Table 3: Query performance (ms). ($K = 10^3, M = 10^6$)

	MC	*FP*	*BiPPR*	*HubPPR*	*BEAR-A*	*BI*
DBLP	11.5K	82.8	19.7	3.1	0.8K	0.1
Web-St	6.1K	0.2K	37.0	8.1	0.1K	0.4
Pokec	0.1M	0.7K	26.9	4.2	-	-
LJ	0.5M	1.0K	59.8	9.1	-	-
Orkut	0.4M	1.4K	0.4K	30.8	-	-
Twitter	2.5M	0.1M	21.5K	3.3K	-	-
UKWeb	2.2M	0.1M	25.9K	3.5K	-	-

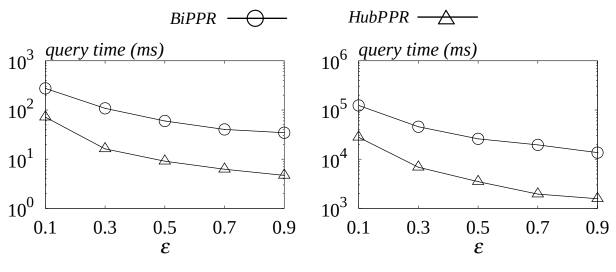
Query sets. We first describe how we generate the query set for PPR queries. For each dataset, we generate 1000 queries with source and target chosen uniformly at random. For top-k PPR queries, there are 12 query sets. The first 5 query sets have $k = 16$, and target set sizes 100, 200, 400, 800, 1600, respectively. The remaining 7 query sets have a common target set size of 400, and varying k values: 1, 2, 4, 8, 16, 32, 64. For each query set, we generate 100 queries, with target nodes chosen uniformly with replacement.

Parameter Setting. Following previous work [20, 21], we set α to 0.2, p_f to $1/n$, and δ to $1/n$. In Section 7.4, we further evaluate the impact of ϵ and the EHI index size to our *HubPPR*. We find that $\epsilon = 0.5$ leads to a good balance between the query accuracy and query performance. Besides, when the index size of *HubPPR* is 5 times the graph size, it strikes a good trade-off between the space consumption and query efficiency. Hence, in the rest of our experiments, we set $\epsilon = 0.5$ and the index size of *HubPPR* to 5 times the graph size. For other competitors, we use their default setting in case that they include additional parameters.

Methods. For the PPR query, we compare *HubPPR* against *BiPPR* [20], *FastPPR* [21], and a baseline Monte-Carlo approach [3]. For indexing methods, we include a version for *BiPPR* (denoted as *BiPPR-I*) that pre-computes and materializes forward and backward phases for all nodes. All of the above methods are implemented in C++, and compiled with full optimizations. Moreover, we compare *HubPPR* against the state-of-the-art index-based solution *BEAR-Approx* [27] with dropout threshold γ (ref. Section 6) set to $1/n$. Note that (i) unlike other methods, *BEAR-Approx* provides no formal guarantee on the accuracy of its results; (ii) *BEAR-Approx* computes the PPR vector from a source node s, while other competitors except for the Monte-Carlo approach computes the PPR score from a source s to a target t. We obtained the binary executables of *BEAR-Approx* from the authors, which is implemented with C++ and Matlab, and compiled with full optimizations. As we will see, *HubPPR* significantly outperforms *Bear-Approx* in terms of index size, query accuracy, query time, and preprocessing time. Among these results, the comparisons on index size and query accuracy are more important due to the fact that *Bear-Approx* is partially implemented with Matlab.

For the top-k PPR query, we evaluate two versions of the proposed top-k PPR algorithm described in Section 4: one without any index, dubbed as *TM* (top-k martingale), and the other leverages the *HubPPR* indexing framework, dubbed as *TM-Hub*. For competitors, we compare with *BEAR-Approx* [27] and the top-k PPR algorithm by Lofgren et al. [20], denoted as *BiPPR-Baseline*. In addition, we compare with a solution that leverages their *BiPPR* algorithm with our *HubPPR* indexing framework, dubbed as *BiPPR-Hub*. We also inspect the accuracy of all methods in comparison. In particular, we report the average recall[2] for each query set with different k values and target sizes. Note that it is computational expensive to derive the exact answer for graphs with billion edges, e.g., UKWeb. As a result, we apply the Monte-Carlo method for each target node in target set T to derive highly accurate results. In particular, we first set the number of random walks to $O(|T| \cdot n \cdot log(n))$, calculate the PPR values for all nodes in T, and select the top-k nodes. Afterwards, we repeatedly double the number of random walks, and recalculate the top-k nodes, until the answer no longer changes.

[2]The precision and recall are the same for top-k PPR queries.



(a) LJ (d) UKWeb

Figure 8: Impact of ϵ to PPR query efficiency.

7.2 PPR Query Efficiency

This section focuses on PPR processing. Table 3 demonstrate the PPR query performance of all solutions on the query set, where *MC*, *FP*, *BEAR-A*, and *BI* are short for *Monte-Carlo*, *FastPPR*, *BEAR-Approx* and *BiPPR-I* algorithms, respectively.

We first inspect the results for index-based approaches, i.e., *HubPPR*, *BiPPR-I* and *BEAR-Approx*. Besides query time, we further report their preprocessing costs in Table 4. Both *BiPPR-I* and *BEAR-Approx* consume an enormous amount of space, and, thus, are only feasible on the smaller DBLP and Web-St datasets under 256GB memory capacity. With the complete materialization of forward and backward search results, *BiPPR-I* achieves the highest query efficiency among all methods. However, the increased performance comes with prohibitive preprocessing costs as shown in Table 4, which renders it inapplicable for large graphs. As *BEAR-Approx* preprocesses the index by constructing sparse sub-matrices, its index size and preprocessing time highly depend on the topology of the input graph. For example, on the DBLP and Web-St datasets, the index size of *BEAR-Approx* is 400x and 10x the original graph, respectively. Meanwhile, the preprocessing time on the DBLP and Web-St datasets are over 24 hours and around 200 seconds, respectively. Compared to the proposed solution *HubPPR*, *BEAR-Approx* consumes over 80x (resp. 2x) space, and yet 250x (resp. 10x) higher query time on DBLP (resp. Web-St). Further, even on the largest dataset UKWeb with 3.7 billion edges, the preprocessing time of *HubPPR* is less than 24 hours on a single machine, which can be further reduced through parallel computation.

Table 4 also demonstrates the compression ratio of our disjoint compression algorithm. The observation is that our disjoint multiset structure is highly effective and it can save up to 99.7% of the space incurred by the list based solution.

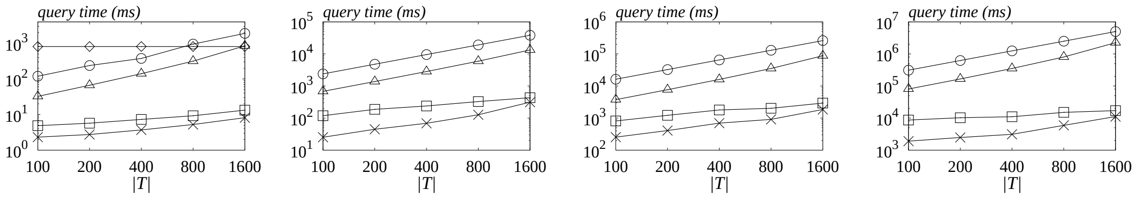
Among all the other methods, *HubPPR* is the most efficient one on all of the tested datasets. In particular, *HubPPR* is 6 to 10 times faster than *BiPPR* with only 5x additional space consumption, which demonstrates the efficiency of our *HubPPR* index. In addition, both *HubPPR* and *BiPPR* are at least one order of magnitude faster than *FastPPR* and the *Monte-Carlo* approach, which is consistent with the experimental result in [20].

In summary, *HubPPR* achieves a good balance between the query efficiency and preprocessing costs, which renders it a preferred choice for PPR queries within practical memory capacity.

7.3 Top-k Query Efficiency and Accuracy

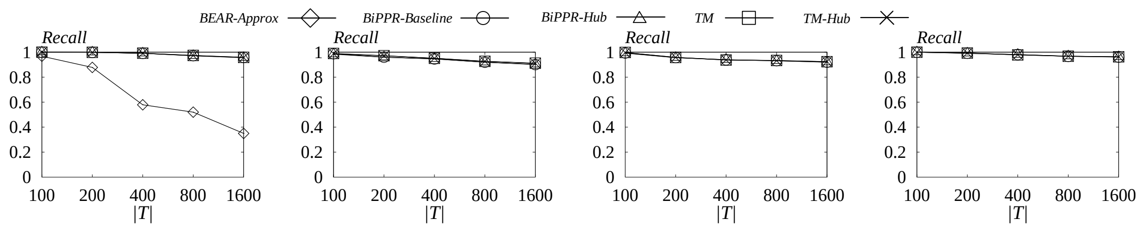
Datasets	**Preprocessing time (sec)**					**Index size**			**Compression ratio**
	HubPPR			*BEAR-Approx*	*BiPPR-I*	*HubPPR*	*BEAR-Approx*	*BiPPR-I*	Disjoint multi-set
	HS	IC	TP						
DBLP	80.4	110.2	190.6	99.4K	8.4K	92.1MB	7.5GB	3.2GB	0.8%
Web-St	18.8	74.9	93.7	175.2	12.7K	51.9MB	113.5MB	6.5GB	0.5%
Pokec	140.5	675.2	815.7	-	-	653.8MB	-	-	2.1%
LJ	419.6	1.5K	1.9K	-	-	1.5GB	-	-	0.8%
Orkut	547.5	6.0K	6.5K	-	-	4.7GB	-	-	2.3%
Twitter	10.9K	30.2K	41.1K	-	-	29.8GB	-	-	0.9%
UKWeb	17.6K	45.1K	62.7K	-	-	77.0GB	-	-	0.3%

Table 4: Preprocessing statistics ($K = 10^3$). The preprocessing time of *HubPPR* includes the time for hub selection (HS), and the time for index construction (IC). The total preprocessing time (TP) of *HubPPR* is the sum of these two costs.



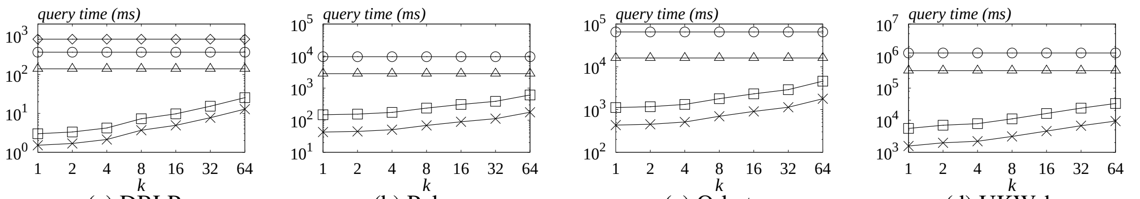
(a) DBLP (b) Pokec (c) Orkut (d) UKWeb

Figure 4: Top-k PPR query efficiency: varying $|T|$.



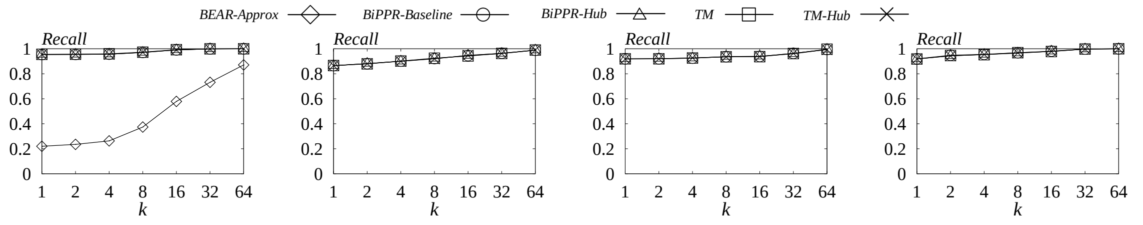
(a) DBLP (b) Pokec (c) Orkut (d) UKWeb

Figure 5: Top-k PPR query accuracy: varying $|T|$.



(a) DBLP (b) Pokec (c) Orkut (d) UKWeb

Figure 6: Top-k PPR query efficiency: varying k.



(a) DBLP (b) Pokec (c) Orkut (d) UKWeb

Figure 7: Top-k PPR query accuracy: varying k.

Next we evaluate top-k PPR processing algorithms. For brevity, we only show the evaluation results on four representative datasets: DBLP, Pokec, Orkut, and UKWeb. Note that *BEAR-Approx* is only feasible on DBLP under 256GB memory capacity among the four datasets.

Figure 4 shows the query efficiency with varying target set size. As we can observe, as the size of the target set T increases, the query latency of *TM*, *TM-Hub*, *BiPPR-Baseline*, and *BiPPR-Hub* all increases. Note that *TM* and *TM-Hub* are less sensitive to the size of T than *BiPPR-Baseline* and *BiPPR-Hub*. For example, on the Orkut dataset, when the size of the target set increases from 100 to 1600, the query latency increases by around 5x for *TM* and *TM-Hub*. In contrast, the query time of *BiPPR-Baseline* and *BiPPR-Hub* increases by around 20x. This is due to our advanced

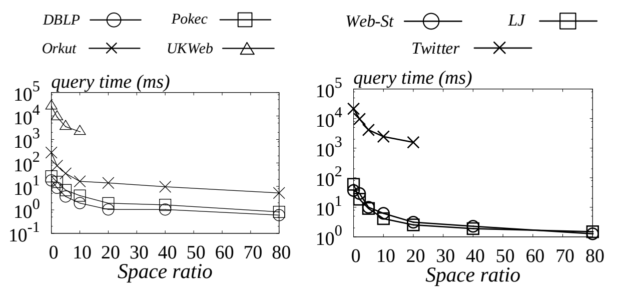


Figure 9: Impact of space to PPR query efficiency.

approximate top-k PPR query algorithm, which iteratively refines the top-k nodes by pruning target nodes that are less likely to be the top-k results.

Specifically, the proposed method *TM-Hub* is up to 220x faster than *BiPPR-Baseline*, and up to 80x faster than *BiPPR-Hub*. For example, on the Pokec dataset, when the size of the target set is 800, *TM-Hub* improves over *BiPPR-Baseline* and *BiPPR-Hub* by 150x and 50x, respectively. Even without any index, *TM* is still up to 90x faster than *BiPPR-Baseline* and 50x faster than *BiPPR-Hub*. In addition, on UKWeb dataset, which includes 3.7 billion edges, our *TM-Hub* can answer the top-k PPR query with target size $|T| = 800$ with only 6 seconds, which demonstrates the efficiency and scalability of our proposed top-k PPR query algorithm and index scheme. Besides, with the proposed *HubPPR* indexing scheme, the query efficiency of both *TM-Hub* and *BiPPR-Hub* are improved several times over their non-index counterparts, which demonstrates the effectiveness of *HubPPR* indexing.

Regarding the query accuracy, *TM*, *TM-Hub*, *BiPPR-Baseline*, and *BiPPR-Hub* all show similarly high recall on the four datasets. This is expected, since (i) the proposed algorithms provide formal and controllable guarantees over the result quality of the top-k PPR query and (ii) *BiPPR-Baseline* also provides approximation guarantee for each PPR value with respect to the source node and an arbitrary target nodes in T, making the selected k nodes of comparably high quality, and (iii) the *HubPPR* indexing framework does not affect result accuracy.

Next, we compare the proposed methods *TM* and *TM-Hub* against *BEAR-Approx*. Note that the query time of *BEAR-Approx* is not affected by the target size $|T|$ since it directly computes the PPR vector for the source node regardless of the targets. Consequently, *BEAR-Approx* incurs unnecessarily hight costs for a small target size $|T|$. For example, when $|T|$ is 100, *TM* and *TM-Hub* are about 160x and 350x faster than *BEAR-Approx*, respectively. Despite the fact that the query performance of *BEAR-Approx* is not affected by the target size $|T|$, the accuracy of *BEAR-Approx* algorithm drops significantly with the increase of $|T|$. In particular, when $|T|$ increases to 1600, the recall drops to 0.35, producing far less accurate query answer than the other four methods. Note that *BEAR-Approx* is only feasible for the DBLP dataset due to its high indexing costs.

Figure 6 shows the evaluation results on top-k PPR query efficiency with varying values of k. Observe that the proposed methods *TM* and *TM-Hub* estimate the top-k PPR queries in an adaptive manner: when k is smaller, both *TM* and *TM-Hub* incur lower query overhead. This behavior is desirable in many real-world applications such as web search. In contrast, the costs for *BiPPR-Baseline* when $k = 1$ and $k = 64$ are the same, which indicates that it incurs a large amount of unnecessary computations for $k = 1$ to derive the top-k PPR queries. The same is true for *BiPPR-Hub* and *BEAR-Approx*. Figure 6 reports the recall for all methods with varying values of k. As expected, *TM*, *TM-Hub*, *BiPPR-Base*, and *BiPPR-Hub* again demonstrate similarly high recall, i.e., over 95% on the majority of datasets when k reaches 8. In contrast, the recall of *BEAR-Approx* drops with decreasing k, and its recall can be as low as around 20%.

In summary, *TM* and *TM-HubPPR* achieves high query efficiency for top-k PPR queries without sacrificing query accuracy, and are adaptive to the choice of k. Their query latency is less sensitive compared to *Bi-PPR-Baseline* with varying target set size. *TM-HubPPR* is often the preferred solution, due to its high query efficiency, formal guarantees on result quality, and moderate space consumption.

7.4 Tuning Parameters

Next we examine the impact of ϵ and index size on the query efficiency of *HubPPR*. For brevity, we only demonstrate the results on PPR query, and omit the results for top-k PPR queries. Figures 8(a) and 8(b) demonstrate the impact of ϵ on PPR query efficiency on LJ and UKWeb datasets, respectively. For completeness, we also include *BiPPR* in the figures. As we can observe, as ϵ decreases, the query efficiency increases for both *HubPPR* and *BiPPR*. In the meantime, the benefit of indexing for *HubPPR* increases with growing ϵ. For example, the improvement of *HubPPR* increases from 4.5x times to 8x over *BiPPR* on the UKWeb dataset when ϵ increases from 0.1 to 0.9. In addition, the effect of ϵ follows a similar trend regardless of the graph size, as shown in the results on the two datasets with very different sizes. We set ϵ to 0.5 since it strikes a good balance between query efficiency and query accuracy.

Finally, Figure 9 shows the impact of the index size on the query efficiency on all the datasets. The x-axis is the ratio of the index size to the input graph size, which we call the *space ratio*. Note that some of the results for Twitter and UKWeb are missing due to limited memory capacity of our testing machine, i.e., 256GB.

As we can observe, the query efficiency of *HubPPR* increases with its index size. This is expected, since a larger index can accommodate more hubs, which is more likely to reduce the cost for both forward search and backward search. The improvement of the query efficiency is more pronounced when the space ratio increases from 1 to 5. For example, on the Web-St dataset, the query efficiency is improved by 10x when the space ratio increases from 1 to 5. On the other hand, the query efficiency grows slowly with the space ratio when the latter becomes larger. In particular, when the space ratio increases from 5 to 80, i.e, 16x index size, the query performance improves by only around 8x. To explain, our forward oracle (resp. backward oracle) iteratively includes the hub that is (estimated to be) the most effective in reducing the cost of random walks (resp. backward propagations). Consequently, as more hubs are included in the index, the marginal benefit in terms of query cost reduction gradually diminishes, since each newly added hub is expected to be less effective in cutting query costs than the ones already selected by the index. We set our index size to 5x of the input graph size, which according to our evaluation results strikes a good tradeoff between the query efficiency and space consumption.

8. CONCLUSION

This paper presents *HubPPR*, an efficient indexing scheme for approximate PPR computation with controllable tradeoffs for accuracy. Our indexing framework includes both a forward oracle and a backward oracle which increase the efficiency of random walks, and the efficiency of the backward propagation, respectively. Meanwhile, we further study how to efficiently answer approximate top-k PPR queries. We present an iterative approach which gradually refines the approximation guarantee for the top-k nodes, and with bounded time returns the desired results. Extensive experiments show that our *HubPPR* improves over existing state-of-the-art PPR query algorithm by 6 to 10 times with only 5 times space consumption. Moreover, our top-k PPR query algorithm improves over the state-of-the-art top-k PPR query algorithm by up to 220 times, which demonstrates the effectiveness of our proposed approximate top-k PPR query algorithm. As future work, we plan to investigate (i) how to devise forward oracle that also considers graph skewness, e.g., allocating different numbers of destinations to different forward hubs; (ii) how to build indices to efficiently process PPR and top-k PPR queries on dynamic graphs; (iii) how to devise effective indexing techniques to improve the PPR vector computation for large graphs without compromising query accuracy.

9. ACKNOWLEDGMENTS

This research is supported by grants MOE2015-T2-2-069 from MOE, Singapore and NPRP9-466-1-103 from QNRF, Qatar.

10. REFERENCES

[1] R. Andersen, C. Borgs, J. T. Chayes, J. E. Hopcroft, V. S. Mirrokni, and S. Teng. Local computation of pagerank contributions. In *WAW*, pages 150–165, 2007.
[2] R. Andersen, F. R. K. Chung, and K. J. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.
[3] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM J. Numerical Analysis*, 45(2):890–904, 2007.
[4] K. Avrachenkov, N. Litvak, D. Nemirovsky, E. Smirnova, and M. Sokol. Quick detection of top-k personalized pagerank lists. In *WAW*, pages 50–61, 2011.
[5] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In *WSDM*, pages 635–644, 2011.
[6] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *SIGMOD*, pages 973–984, 2011.
[7] P. Berkhin. Survey: A survey on pagerank computing. *Internet Mathematics*, 2(1):73–120, 2005.
[8] P. Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.
[9] S. Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW*, pages 571–580, 2007.
[10] F. R. K. Chung and L. Lu. Survey: Concentration inequalities and martingale inequalities: A survey. *Internet Mathematics*, 3(1):79–127, 2006.
[11] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.
[12] Y. Fujiwara, M. Nakatsuji, H. Shiokawa, T. Mishima, and M. Onizuka. Efficient ad-hoc search for personalized pagerank. In *SIGMOD 2013*, pages 445–456, 2013.
[13] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka. Efficient personalized pagerank with accuracy assurance. In *KDD*, pages 15–23, 2012.
[14] F. L. Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303, 2014.
[15] M. S. Gupta, A. Pathak, and S. Chakrabarti. Fast algorithms for topk personalized pagerank queries. In *WWW*, pages 1225–1226, 2008.
[16] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *WWW*, pages 505–514, 2013.
[17] G. Iván and V. Grolmusz. When the web meets the cell: using personalized pagerank for analyzing protein interaction networks. *Bioinformatics*, 27(3):405–407, 2011.
[18] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.
[19] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.
[20] P. Lofgren, S. Banerjee, and A. Goel. Personalized pagerank estimation and search: A bidirectional approach. In *WSDM*, pages 163–172, 2016.
[21] P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *KDD*, pages 1436–1445, 2014.
[22] T. Maehara, T. Akiba, Y. Iwata, and K.-i. Kawarabayashi. Computing personalized pagerank quickly by exploiting graph structures. *PVLDB*, 7(12):1023–1034, 2014.
[23] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis. Frogwild! - fast pagerank approximations on graph engines. *PVLDB*, 8(8):874–885, 2015.
[24] R. Motwani and P. Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.
[25] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.
[26] A. D. Sarma, A. R. Molla, G. Pandurangan, and E. Upfal. Fast distributed pagerank computation. In *ICDCN*, pages 11–26, 2013.
[27] K. Shin, J. Jung, L. Sael, and U. Kang. BEAR: block elimination approach for random walk with restart on large graphs. In *SIGMOD 2015*, pages 1571–1585, 2015.
[28] D. Williams. *Probability with martingales*. Cambridge university press, 1991.
[29] F. Zhu, Y. Fang, K. C. Chang, and J. Ying. Incremental and accuracy-aware personalized pagerank through scheduled approximation. *PVLDB*, 6(6):481–492, 2013.

APPENDIX

Proof of Lemma 2. From the definition of backward search, if an unit information is propagated from u, then eventually $\pi(s,u)$ is propagated to node s. By scaling it with τ, it can be derived that, if τ information is propagated from u, eventually $\tau \cdot \pi(s,u)$ is propagated to s. As a result, we have the following equation.

$$\pi(s,u) = \frac{1}{\tau} \cdot \pi^{\dashv}(s,u,\tau) + \frac{1}{\tau} \cdot \sum_{v \in V} \pi(s,v) \cdot r(v,u,\tau).$$

Then by applying Equation 3 with the current residue and reserve states, and replacing $\pi(s,u)$ with the above equation, we have

$$\pi(s,t) = \sum_{v \in V \setminus \{u\}} \pi(s,v) \cdot (r(v,t) + \frac{r(u,t)}{\tau} \cdot r(v,u,\tau)) + \pi(s,u) \cdot \frac{r(u,t)}{\tau} \cdot r(u,u,\tau) + (\pi^{\dashv}(s,t) + \frac{r(u,t)}{\tau}\pi^{\dashv}(s,u,\tau)) \tag{9}$$

Equation 9 exactly describes the property of Lemma 2, which finishes the proof. □

Proof of Lemma 3. First, we prove that if $(1+\epsilon) \cdot LB(t_i) \geq UB(t_i)$, then

$$(1-\epsilon/2) \cdot \pi(s,t_i) \leq \hat{\pi}(s,t_i) \leq (1+\epsilon/2) \cdot \pi(s,t_i). \tag{10}$$

As $\hat{\pi}(s,t_i) = (LB(t_i) + UB(t_i))/2$, it can be derived that:

$$\begin{aligned}\hat{\pi}(s,t_i) &\leq (LB(t_i) + (1+\epsilon) \cdot LB(t_i))/2 \\ &= (1+\epsilon/2) \cdot LB(t_i) \\ &\leq (1+\epsilon/2) \cdot \pi(s,t_i).\end{aligned}$$

At the meantime,

$$\begin{aligned}\hat{\pi}(s,t_i) &\geq (\frac{1}{1+\epsilon} \cdot UB(t_i) + UB(t_i))/2 \\ &\geq ((1-\epsilon)+1)/2 \cdot UB(t_i) \\ &\geq (1-\epsilon/2) \cdot \pi(s,t_i).\end{aligned}$$

This finishes the proof for Equation 10. Next we prove that if $(1+\epsilon)\cdot LB(t_i) \geq UB(t_i)$, then

$$(1-\epsilon)\cdot\pi(s,t_i^*) \leq \hat{\pi}(s,t_i) \leq (1+\epsilon)\cdot\pi(s,t_i^*). \quad (11)$$

We first have the following observation: Let t_i (resp. t_i') be the node with the i-th largest PPR lower (resp. upper) bound in T. If $(1+\epsilon)\cdot LB(t_i) \geq UB(t_i)$, then $(1+\epsilon)\cdot LB(t_i) \geq UB(t_i')$.

To prove, we first obtain two ordered sequences. One sequence $\mathcal{S}_1$ is $LB(t_1)$, $LB(t_2), \cdots, LB(t_k)$, and the other one $\mathcal{S}_2$ is $UB(t_1), UB(t_2), \cdots, UB(t_k)$. We then apply the bubble sort on $\mathcal{S}_2$ in decreasing order of the element values in $\mathcal{S}_2$, in which case $\mathcal{S}_2$ finally becomes $UB(t_1'), UB(t_2'), \cdots, UB(t_k')$. We consider the gap ratio between the i-th item in $\mathcal{S}_1(i)$ and $\mathcal{S}_2(i)$. Initially, it can be guaranteed that $\mathcal{S}_2(i)/\mathcal{S}_1(i) \leq 1+\epsilon$ for all $1 \leq i \leq k$. Our main target is to show that through the sorting process, it always guarantee that $\mathcal{S}_2(i)/\mathcal{S}_1(i) \leq 1+\epsilon$. We prove by induction. Initially, it is clear that $\mathcal{S}_2(i)/\mathcal{S}_1(i) \leq 1+\epsilon$. Then, if two elements, say $\mathcal{S}_2(j)$ and $\mathcal{S}_2(j+1)$ are swapped, then we know that $\mathcal{S}_2(j) < \mathcal{S}_2(j+1)$. Meanwhile, $\mathcal{S}_1(j) > \mathcal{S}_1(j+1)$ from its definition. Then

$$\frac{\mathcal{S}_2(j)}{\mathcal{S}_1(j+1)} \leq \frac{\mathcal{S}_2(j+1)}{\mathcal{S}_1(j+1)} \leq 1+\epsilon.$$
$$\frac{\mathcal{S}_2(j+1)}{\mathcal{S}_1(j)} \leq \frac{\mathcal{S}_2(j+1)}{\mathcal{S}_1(j+1)} \leq 1+\epsilon. \quad (12)$$

Hence, after the swapping of elements in $\mathcal{S}_2$, it is still guaranteed that $\mathcal{S}_2(i)/\mathcal{S}_1(i) \leq 1+\epsilon$. By induction, it can be proved that when the sorting finishes, $\mathcal{S}_2(i)/\mathcal{S}_1(i) \leq 1+\epsilon$ still holds, i.e., $(1+\epsilon)\cdot LB(t_i) \geq UB(t_i')$.

Let t_i^* be the node with the i-th highest PPR value. It is clear that $LB(t_i) \leq \pi(s,t_i^*) \leq UB(t_i')$, $LB(t_i) \leq \pi(s,t_i) \leq UB(t_i)$. Then,

$$\pi(s,t_i) \leq UB(t_i) \leq LB(t_i)\cdot(1+\epsilon) \leq \pi(s,t_i^*)\cdot(1+\epsilon).$$

Meanwhile,

$$\pi(s,t_i) \geq LB(t_i) \geq \frac{1}{1+\epsilon}\cdot UB(t_i')$$
$$\geq (1-\epsilon)\cdot UB(t_i') \geq (1-\epsilon)\cdot\pi(s,t_i^*).$$

This finishes the proof.

□

Proof of Lemma 5. First note that $\lambda = \sqrt{\left(\frac{2M}{3}\ln p_f^*\right)^2 - 2b\cdot\ln p_f^*} - \frac{2M}{3}\ln p_f^*$ is the root of equation $\lambda^2 + 2b\cdot\ln p_f^* + \frac{2M\cdot\lambda}{3}\cdot\ln p_f^*$=0. Then apply Theorem 4, we have that with probability p_f^*:

$$\Pr[|M_{\Omega_i} - (\pi(s,t) - \pi^{\dashv}(s,t))\cdot\Omega_i| \geq \lambda] \leq \exp\left(-\frac{\lambda^2}{2b + 2M\cdot\lambda/3}\right)$$
$$= \exp(\ln p_f^*) = p_f^*$$

As a result, we have $\frac{M_{\Omega_i}-\lambda}{\Omega_i} \leq \pi(s,t) - \pi^{\dashv}(s,t) \leq \frac{M_{\Omega_i}+\lambda}{\Omega_i}$. Moreover, as $0 \leq \pi(s,t) \leq 1$, we have the desired bound, which finishes the proof of Lemma 5. □

Proof of Corollary 1. Consider the backward search cost for each target node. As the worst case running time for the backward search from a target node t is $\Theta(n)$. We use n^2 as a upper bound for the backward search cost. Then, the total number of sampled random walks is bounded by $n^2\cdot\alpha$. As a result, our algorithm invokes Lemma 5 at most $\log(n^2\cdot\alpha\cdot|T|)\cdot|T|$ times.

By applying union bound, we have: the probability that all bounds are correct bounds is no less than $1 - p_f^*\cdot\log(n^2\cdot\alpha\cdot|T|)\cdot|T| = 1 - \frac{p_f}{2}$, which finishes the proof. □

Proof of Lemma 6. Suppose either $\omega \geq 2\cdot\frac{r_{max}}{\epsilon^2\cdot\delta}\cdot\log\frac{2k}{p_f}$ or $r_{max}(t) \leq r_{max}$ does not hold for all target nodes, and Algorithm 2 terminates, then it terminates at Line 15. Hence, if we can prove that when $\omega \geq 2\cdot\frac{r_{max}}{\epsilon^2\cdot\delta}\cdot\log\frac{2k}{p_f}$ and $r_{max}(t) \leq r_{max}$, it has at least $1-p_f/2$ probability that it terminates at Line 15, then Algorithm 2 has at least $1-p_f/2$ probability that it terminates at Line 15.

Consider the last iteration of our top-k PPR query algorithm. Based on how random walks are generated, it can be verified that half of the total sampled random walks are generated in the last iteration. As a result, in the last iteration, it generates at least $\omega/2 = \frac{r_{max}}{\epsilon^2\cdot\delta}\cdot\log\frac{2k}{p_f}$ random walks. In addition, for any target node t in T, it satisfies that $r_{max}(t) \leq r_{max}$. In addition, we have

$$\Pr[(1+\epsilon)\cdot LB(t_i) \geq UB(t_i)] \geq$$
$$\Pr[|\pi(s,t_i^*) - \pi(s,t_i)| \leq \epsilon\pi(s,t_i^*)].$$

Then, based on the correctness of *BiPPR*, we have: with probability of $1-\frac{p_f}{2k}$, for an arbitrary i ($1 \leq i \leq k$), the i-th estimation satisfies the approximation ratio, i.e., satisfying Line 14. Then by applying the union bound on the top-k nodes, the algorithm will terminates at Line 15 with at least $1-\frac{p_f}{2k}*k = 1-p_f/2$ probability, which finishes the proof. □

Proof of Theorem 3. Based on Lemma 5, Lemma 6, and Corollary 1, we can derive that the probability that the returned top-k nodes bears at most ϵ relative error is no more than $1-2\cdot p_f/2 = 1-p_f$, which finishes the proof of the approximation guarantee.

Next, consider the time complexity of our top-k PPR query algorithm. For the forward phase, our algorithm samples at most $\frac{2r_{max}}{\epsilon^2\cdot\delta}\cdot\log\frac{2k}{p_f}$ random walks, which has a cost of $O\left(\frac{r_{max}}{\alpha\epsilon^2\cdot\delta}\cdot\log\frac{k}{p_f}\right)$. Besides, as the average running time for a single backward search with a random target is $O\left(\frac{m}{n\cdot\alpha\cdot r_{max}}\right)$ [20], then the average running time of the backward phase is $O\left(\frac{m\cdot|T|}{n\cdot\delta\cdot\alpha}\right)$, since it proceeds at most $|T|$ backward searches. As a result, the average running time of our top-k PPR query algorithm is $O\left(\frac{r_{max}}{\alpha\epsilon^2\cdot\delta}\cdot\log\frac{k}{p_f} + \frac{m\cdot|T|}{n\cdot\alpha\cdot r_{max}}\right)$. By setting $r_{max} = \sqrt{\frac{\epsilon^2\cdot\delta\cdot m\cdot|T|}{n\cdot\log(k/p_f)}}$, we have the desired running time complexity $O\left(\frac{1}{\alpha\cdot\varepsilon}\sqrt{\frac{m\cdot|T|}{n\delta}\log\frac{k}{p_f}}\right)$. □

Proof of Lemma 7. When a snapshot $S(h_b,\tau)$ is used to speedup the backward search, the cost of the updating is at most $|S(h_b,\tau)|$. At the meantime, notice that if the initial residual from h_b is τ' before the update, then the cost of the backward search from h_b with message size τ' without index is no smaller than the backward search cost with τ_i ($\tau_i \leq \tau' < \tau_{i-1}$), where τ_i is the initial residual of the i-th snapshot in $\mathcal{B}(h_b)$. Next, we consider the cost of the update with snapshot τ_i. Since if a node has non-zero reserve / residue, the node would be at least involved in a backward propagation, then the update cost is no more than twice the backward search cost. Meanwhile, given an initial message τ', the snapshot that is to be chosen is at most twice as large as $|S(h_b,\tau_i)|$, indicating the update cost is bounded by the backward search cost. As a consequence, the backward search with $\mathcal{B}$ has the similar cost as the backward phase in *BiPPR*. In summary, the amortized cost of our backward search with backward oracle is identical to the case without index, which is $O(\frac{m}{n\cdot\alpha\cdot r_{max}})$. □