Efficient Algorithms for Finding Approximate Heavy Hitters in Personalized PageRanks

Sibo Wang
University of Queensland
Brisbane, Australia
sibo.wang@uq.edu.au

Yufei Tao
Chinese University of Hong Kong
New Territories, Hong Kong
taoyf@cse.cuhk.edu.hk

ABSTRACT

Given a directed graph G, a source node s, and a target node t, the *personalized PageRank (PPR)* of t with respect to s is the probability that a random walk starting from s terminates at t. The average of the personalized PageRank score of t with respect to each source node $v \in V$ is exactly the PageRank score $\pi(t)$ of node t, which denotes the overall importance of node t in the graph. A *heavy hitter* of node t is a node whose contribution to $\pi(t)$ is above a ϕ fraction, where ϕ is a value between 0 and 1. Finding heavy hitters has important applications in link spam detection, classification of web pages, and friend recommendations.

In this paper, we propose *BLOG*, an efficient framework for three types of heavy hitter queries: *the pairwise approximate heavy hitter (AHH)*, *the reverse AHH*, and *the multi-source reverse AHH* queries. For pairwise AHH queries, our algorithm combines the Monte-Carlo approach and the backward propagation approach to reduce the cost of both methods, and incorporates new techniques to deal with high in-degree nodes. For reverse AHH and multi-source reverse AHH queries, our algorithm extends the ideas behind the pairwise AHH algorithm with a new "logarithmic bucketing" technique to improve the query efficiency. Extensive experiments demonstrate that our BLOG is far more efficient than alternative solutions on the three queries.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**;

KEYWORDS

Heavy Hitters; Personalized PageRank; Approximate Algorithms; Social Recommendation

ACM Reference Format:
Sibo Wang and Yufei Tao. 2018. Efficient Algorithms for Finding Approximate Heavy Hitters in Personalized PageRanks. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3196919

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196919

1 INTRODUCTION

Let $G = (V, E)$ be a directed graph, and define $n = |V|$. Given a source node s, and a target node t, the *personalized PageRank (PPR)* of t with respect to s, denoted as $\pi(s, t)$, is the probability that a random walk starting from s terminates at t. Define $\pi(t) = \sum_{s \in V} \pi(s, t)$; $\pi(t)$ is exactly n times the PageRank of t.

This work was motivated by an interesting observation: a large $\pi(s, t)$ implies that s and t are *mutually* important to each other:

- The first direction is straightforward: a higher $\pi(s, t)$ suggests greater importance of t to s, as is the rationale behind PPR in the first place.
- The other direction is more subtle: a higher $\pi(s, t)$ indicates that s makes a larger contribution to the PageRank of t (i.e., $n \cdot \pi(t)$), thus increasing the importance of s to t as well.

Although PPRs have been widely applied in social networking services to make friend recommendations, this is usually done by leveraging only the first "direction of importance". To illustrate, consider the standard setup where nodes correspond to users and edges correspond to friendship relationships. Upon detecting that $\pi(s, t)$ is high and yet t is not a friend of s, a social networking service (e.g., LinkedIn[1]) would recommend t to s. The issue, however, is that t may simply ignore the request from s — resulting in a "silence after request" situation — if t does not consider s important.

The second "direction of importance" offers a promising approach to enhance the effectiveness of recommendations. Continuing the above setup, if $\pi(s, t)$ also makes a heavy contribution to $\pi(t)$, the chance of t willing to accept s naturally increases. Thus, by taking both directions into account, we would be able to avoid at least some of the "silence after request" encountered previously.

We introduce a new notion called "PageRank heavy hitter" to quantify importance of the second direction, and thereby, gives a convenient way to harness this direction for recommendation. Specifically, s is said to be a *ϕ-heavy hitter* of t if $\pi(s, t) > \phi \cdot \pi(t)$, where ϕ is a real number satisfying $0 < \phi < 1$. Accordingly, a social networking site would now recommend s to t only when (i) $\pi(s, t)$ is high enough, and (ii) s is a ϕ-heavy hitter of t for an appropriate ϕ. It is worth mentioning that, although we have used social networking as an example, similar recommendation operations are also crucial in other applications like link spam detection [6, 16] and classification of web pages [17]. Our proposition is meaningful in those applications as well.

However, discovering heavy hitters is non-trivial, partly because PPRs — which underlie heavy hitters — are very expensive to compute precisely with the algorithms known so far. On the other

[1] See https://databricks.com/session/random-walks-on-large-scale-graphs-with-apache-spark

hand, significant progress has been made in recent years on approximating PPR values with small errors. This creates hopes that certain approximation variants of heavy hitters could also be extracted efficiently. We will show that this is indeed the case when one is willing to relax ϕ by a constant factor $0 < c < 1$. This gives rise to *c-approximate heavy hitter*. Defined formally in Section 2.1, this notion intuitively says that, for the purpose of finding ϕ-heavy hitters of t approximately, a node s must (i) be reported if $\pi(s,t) > (1+c)\cdot\phi\cdot\pi(t)$, and (ii) *not* be reported when $\pi(s,t) < (1-c)\cdot\phi\cdot\pi(t)$. Created in between is a "don't-care" case when $\pi(s,t)$ is in $[(1-c)\phi\cdot\pi(t), (1+c)\phi\cdot\pi(t)]$; it is this relaxation that brings about significant savings in the computation cost.

Contributions. We propose *BLOG*[2], an efficient framework for processing three types of approximate heavy hitter (AHH) queries, which can be intuitively understood as follows (formal definitions will appear in Section 2.1):

- Pairwise AHH query: Given a pair of nodes s and t, decide whether s is a c-approximate heavy hitter of t.
- Reverse AHH query: Given a node s and a set T of nodes, find all nodes $t \in T$ such that s is a c-approximate heavy hitter of t, possibly ignoring the don't-care case.
- Multi-source reverse AHH query: Given two sets S, T of nodes, the query answers, for each $s \in S$, a reverse AHH query with respect to s and T.

Note that the pairwise AHH query is a special case of the reverse AHH query, which in turn is subsumed by the multi-source reverse AHH query. They correspond to different recommendation scenarios: specifically, "point-to-point" (e.g., introducing Alice to Bob?), "one-to-many" (e.g., for Alice, make recommendations from the people in her company), and "many-to-many" (e.g., make recommendations between people in the database area, and those in networking), respectively.

Although PageRank heavy hitter is a new concept, in Section 2.2 we show that several existing algorithms [1, 3] designed for approximate PPR computation can be adapted to answer AHH queries as well. Our technical novelty, however, lies in leveraging the specific properties of AHH to improve the computational complexities of those adapted algorithms:

- In Section 3, We combine the Monte-Carlo approach of [3] and the backward propagation algorithm of [1] to obtain our first algorithm for the pairwise AHH query. This algorithm is in spirit similar to the bidirectional approach of [22] (for PPR computation), but incorporates analysis tailored made for AHHs.
- In Section 4, we enhance our pairwise-AHH algorithm with new techniques for handling high in-degree nodes. Those techniques improve the algorithm's time efficiency without compromising its approximation guarantees.
- In Sections 5 and 6, we describe algorithms for answering reverse AHH and multi-source reverse AHH queries, respectively. These algorithms are built upon the ideas used to solve pairwise AHH queries, but in addition deploy a new technique we call "logarithmic bucketing" that significantly reduces the time complexities.

[2] Bidirectional heavy hitter with Logarithmic bucketing.

  Notice that a reverse AHH query can be reduced to several pairwise AHH queries, and similarly, a multi-source reverse AHH query to several reverse AHH queries. Our algorithms in Sections 5 and 6 improve over those simple reductions; in order words, better time complexities can be acquired by utilizing the characteristics of each individual problem.

After surveying the related work in Section 7, we in Section 8 experimentally compare the proposed BLOG method against alternative solutions using 6 real large datasets with up to 1.5 billion edges. The results demonstrate that our solutions outperform the competitors by a factor up to orders of magnitude under the same approximation guarantees. Finally, Section 9 concludes the paper with a summary of our findings.

2 PRELIMINARIES

2.1 Problem Definition

Let $G = (V, E)$ be the input graph, and set $n = |V|$. Given a source node s, and a decay factor α, a *random walk* from s is a traversal of graph G from s such that at each step, it either stops at the current node with probability α, or randomly jumps to an out-neighbour of the current node.

Definition 2.1. The *personalized PageRank* (PPR) $\pi(s,t)$ of node t with respect to s is the probability that a random walk from s terminates at t. □

The value of $\pi(s,t)$ reflects the importance of t from the viewpoint of s (i.e., the first direction of importance explained in Section 1).

Definition 2.2. The *PageRank* of node t is $\frac{1}{n}\sum_{s\in V}\pi(s,t)$, i.e., the average personalized PageRank of t with respect to all $s \in V$. □

The PageRank of t indicates the overall importance of node t in the graph. Define $\pi(t) = \sum_{s\in V}\pi(s,t)$. Note that $\pi(t)$ scales up the PageRank of t by a factor of n, and is in the range of $[0, n]$.

Definition 2.3 (Heavy hitter). Given a real value $0 < \phi < 1$, we say that a node $s \in V$ is a *ϕ-heavy hitter* of a node $t \in V$ if $\pi(s,t) > \phi\cdot\pi(t)$. □

We will consider only $\phi\cdot\pi(t) \le 1$; otherwise t has no heavy hitters, because $\pi(s,t) \le 1$ for any $s \in V$. The smallest ϕ at which s is a ϕ-heavy hitter of t reflects the importance of s to t (i.e., the first direction of importance explained in Section 1).

Deriving ϕ-heavy hitters would require the exact PPR values, which are expensive to compute. We instead work with $O(1)$-approximate heavy hitters defined as:

Definition 2.4 (c-approximate heavy hitter). Given a real value $0 < \phi < 1$, a constant real value $0 < c < 1$, two nodes s, t in V, we say that s is:

- a *c-absolute ϕ-heavy hitter* of t if $\pi(s,t) > (1+c)\phi\cdot\pi(t)$;
- a *c-permissible ϕ-heavy hitter* of t if $(1-c)\phi\cdot\pi(t) \le \pi(s,t) \le (1+c)\phi\cdot\pi(t)$;
- not a c-approximate ϕ-heavy hitter of t, otherwise. □

Example 2.5. Given three source nodes s_1, s_2, and s_3, a target node t, assume that $\pi(s_1,t) = 0.026, \pi(s_2,t) = 0.021$, $\pi(s_3,t) = 0.018$, and $\pi(t) = 0.23$. Let $\phi = 0.1$. Since $\pi(s_1,t) = 0.026 > \phi\cdot$

$\pi(t) = 0.1 \cdot 0.23 = 0.023$, s_1 is a ϕ-heavy hitter of t. Similarly, we can verify that neither s_2 nor s_3 is a ϕ-heavy hitter of t. Set $c = 0.1$. Since $\pi(s_1, t) = 0.026 > (1+c)\cdot\phi\cdot\pi(t) = (1+0.1)\cdot 0.1\cdot 0.23 = 0.0253$, s_1 is a c-absolute ϕ-heavy hitter of t. For s_2, $\pi(s_2, t) = 0.021 \leq (1+c)\phi\cdot\pi(t) = 0.0253$, and $\pi(s_2, t) = 0.021 \geq (1-c)\cdot\phi\cdot\pi(t) = 0.0207$; hence, s_2 is a c-permissable ϕ-heavy hitter of t. For s_3, since $\pi(s_3, t) = 0.018 < (1-c)\cdot\phi\cdot\pi(t) = 0.0207$, s_3 is not a c-approximate ϕ-heavy hitter of t. □

Next, we formalize the three types of approximate heavy hitter queries to be studied, in ascending order of generality:

Definition 2.6 (Pairwise approximate heavy hitter (AHH) query). Given a source node $s \in V$, a target node $t \in V$, a real value $0 < \phi < 1$, and a constant real value $0 < c < 1$, a *pairwise AHH* query returns:

- true, if s is c-absolute ϕ-heavy hitter of t;
- either true or false, if s is c-permissible ϕ-heavy hitter of t;
- false, otherwise. □

Definition 2.7 (Reverse AHH query). Given a source node $s \in V$, a target set $T \subseteq V$, a real value $0 < \phi < 1$, and a constant real value $0 < c < 1$, a *reverse AHH* query returns a set $H \subseteq T$ such that, for each node $t \in T$:

- if s is a c-absolute ϕ-heavy hitter of t, t must belong to H;
- if s is a c-permissible ϕ-heavy hitter of t, t may or may not belong to H;
- otherwise, t must not belong to H. □

Definition 2.8 (Multi-source Reverse AHH query). Given a source set $S \subseteq V$, a target set $T \subseteq$, a real value $0 < \phi < 1$, and a constant real value $0 < c < 1$, a *multi-source reverse AHH query* returns, for each node $s \in S$, a set $H_s \subseteq T$ such that for each node $t \in T$:

- if s is a c-absolute ϕ-heavy hitter of t, t must belong to H_s;
- if s is a c-permissable ϕ-heavy hitter of t, t may or may not belong to H_s;
- otherwise, t must not belong to H_s. □

In all three queries, ϕ is not fixed in advance, but instead, is supplied at query time. This makes it infeasible to pre-compute the results for all possible values of ϕ. Another straightforward solution is to store the PPR value $\pi(s, t)$ for all $(s, t) \in V \times V$, but its $\Theta(n^2)$ space overhead is prohibitively expensive.

We, on the other hand, permit an algorithm to store the PageRank value of each node $t \in V$ (equivalently, $\pi(t)$), which requires only $O(n)$ space. These PageRank values can be computed with the Power-Iteration method [26] in a pre-processing step. We also require that the algorithm must be able to answer each query correctly with a probability at least $1 - 1/n$.

Table 1 lists the notations frequently used in the paper (some notations will appear in later sections).

Abbreviation Conventions. In subsequent discussions, we will almost always concentrate on one specific query, whose values of ϕ and c will be clear from the context, and remain the same throughout the discussion. When this is true, we will refer to a c-absolute ϕ-heavy hitter simply as an "absolute AHH", and a c-permissible ϕ-heavy hitter as a "permissible AHH". Furthermore, by "a node s is an AHH of node t", we mean that s is either an absolute or permissible AHH of t; likewise, by "s is not an AHH of t", we mean that s is neither an absolute AHH nor a permissible AHH of t.

Notation	Description
G=(V, E)	The input graph G with node set V and edge set E
n, m	The number of nodes and edges in G, respectively
$Out(v)$	The set of out-neighbors of node v
$In(v)$	The set of in-neighbors of node v
$\pi(s, t)$	The exact PPR value of t with respect to s
α	The decay factor in the random walk
ϕ	The fraction threshold in heavy hitter definition (Ref. Definition 2.3)
c	The approximation factor in AHH definition (Ref. Definition 2.4)
r_{max}	The threshold to terminate backward propagation
d^-_{max}	The maximum in-degree
$\boldsymbol{r}$, $\boldsymbol{p}$	The two vectors maintained in the backward propagation (Ref. Section 2.2 for the detailed definition)
T	The target set in the reverse AHH and multi-source reverse AHH query
S	The source set in the multi-source reverse AHH query

Table 1: Frequently used notations.

2.2 Adapting Existing Solutions

Monte Carlo. The classic solution for computing PPRs is the Monte-Carlo approach. Given a source node s, the approach samples ω random walks, records the number $c(t)$ of random walks that terminate at t, and uses $\hat{\pi}(s, t) = \frac{c(t)}{\omega}$ as an estimate of $\pi(s, t)$. For the Monte-Carlo approach, we have the following lemma[3]:

Lemma 2.9 (Monte-Carlo). *Let $\lambda > 0$ be an absolute error threshold. When $\omega = (2+\lambda)\cdot\frac{\log(1/p_f)}{\lambda^2}$, the Monte-Carlo approach guarantees that $|\hat{\pi}(s, t) - \pi(s, t)| \leq \lambda$ holds with $1 - p_f$ probability.*

As a result, by setting $\lambda = c\cdot\phi\cdot\pi(t)$ and $p_f = 1/n$, we can answer the pairwise AHH query with $O\left(\frac{\log n}{(c\cdot\phi\cdot\pi(t))^2}\right)$ random walks. Given a source node s and a target set T, define $\pi_{min} = \min_{t\in T}\pi(t)$. Then, by sampling $\omega = O\left(\frac{\log n}{(c\cdot\phi\cdot\pi_{min})^2}\right)$ random walks from s, the Monte-Carlo approach is able to answer the AHH query from s to t, for any $t \in T$. Therefore, it also supports the reverse AHH query. To support multi-source reverse AHH queries, it samples the same number of random walks from each node s in the source set S.

Backward Propagation. The backward propagation algorithm is proposed by Andersen et al. [1]. Instead of starting from the source s, it starts from the target node t, and derives an estimation $\hat{\pi}(v, t)$ for each node $v \in V$. The main idea is that given two nodes s and t, the personalized PageRank $\pi(s, t)$ satisfies the following equation:

$$\pi(s,t) = \sum_{v\in In(t)} \frac{(1-\alpha)\cdot\pi(s,v)}{|Out(v)|} + \begin{cases}\alpha, \text{ if } s = t,\\ 0, \text{ otherwise.}\end{cases} \quad (1)$$

[3] All the omitted proofs can be found in Appendix A.

Algorithm 1: Backward Propagation

Input: Graph G, target node t, probability α, residue threshold r_{max}
Output: $\boldsymbol{p}$, $\boldsymbol{r}$

1 $\boldsymbol{r}(t) \leftarrow 1$; $\boldsymbol{r}(v) \leftarrow 0$ for all $v \neq s$;
2 $\boldsymbol{p} \leftarrow \boldsymbol{0}$;
3 **while** $\exists v \in V$ *such that* $\boldsymbol{r}(v) > r_{max}$ **do**
4 $\boldsymbol{p}(v) += \alpha \cdot \boldsymbol{r}(v)$;
5 $r' = \boldsymbol{r}(v)$; $\boldsymbol{r}(v) = 0$;
6 **for** *each* $u \in In(v)$ **do**
7 $\boldsymbol{r}(u) += r' \cdot \frac{1-\alpha}{|Out(u)|}$;

The proof of this equation is shown in Appendix A. The equation indicates the relationships of the PPRs of different nodes with respect to s, and we omit the source s in the following explanation. The equation shows that the PPR of t can be represented with the sum of the PPR of each $v \in In(t)$, where $In(t)$ is the set of in-neighbours of t. Similarly, the PPR of v, where v is an in-neighbor of node t, can be further represented with the sum of PPRs of the in-neighbors of v. At the end, the PPR $\pi(s,t)$ can be represented by the sum of the PPRs of all nodes that can reach t with different coefficients and a constant related to the source s. That is the main intuition of the backward propagation. The pseudo-code of the backward propagation is shown in Algorithm 1. The algorithm maintains two vectors $\boldsymbol{p}$ and $\boldsymbol{r}$, where $\boldsymbol{r}$ maintains the coefficient part and $\boldsymbol{p}$ maintains the constant part for each source s. Initially, it starts with $\boldsymbol{p} = \boldsymbol{0}$ and $\boldsymbol{r}(v) = 0$ for all $v \neq t$ while $\boldsymbol{r}(t) = 1$ (Algorithm 1 Lines 1-2). Then, the backward propagation applies a series of *pushback* operations while maintaining that the following equation holds for an arbitrary node s:

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

The pushback operation adjusts the coefficient part $\boldsymbol{r}$ and the constant part $\boldsymbol{p}$ based on Equation 1. The details of how they are adjusted is as shown in Algorithm 1 Lines 4-7. The backward propagation terminates when for any node $v \in V$, $\boldsymbol{r}(v)$ is no larger than r_{max} (Algorithm 1 Line 3). It is also proven in [1] that, when the backward propagation finishes, the total number of pushback operations can be bounded by $\frac{\pi(t)}{\alpha \cdot r_{max}}$. Therefore, the backward propagation algorithm has a time complexity of $O\left(\frac{\pi(t) \cdot c_{push}}{r_{max}}\right)$, where c_{push} is the cost of a pushback operation.

Notice that when the backward propagation finishes, it holds that for any node v, $r(v) < r_{max}$. Combining with Equation 2, for any node s, it holds that $\boldsymbol{p}(s) \leq \pi(s,t) \leq \boldsymbol{p}(s) + r_{max}$. Then, by setting $r_{max} = c \cdot \phi \cdot \pi(t)$, we can answer pairwise AHH queries, with a time complexity of $O\left(\frac{c_{push}}{c \cdot \phi \cdot r_{max}}\right)$. To answer reverse AHH queries or multi-source reverse AHH queries, we simply apply the backward propagation algorithm from each target node $t \in T$.

3 PAIRWISE APPROXIMATE HEAVY HITTER

For pairwise AHH query, we first present an improved version of the Monte-Carlo approach in Section 3.1. Next, inspired by BiPPR [22], we apply a bidirectional approach to solve the pairwise approximate heavy hitter query in Section 3.2.

3.1 A Tighter Bound with Monte-Carlo

Recall that to solve the pairwise AHH query, the classic Monte-Carlo approach works by setting $\lambda = c \cdot \phi \cdot \pi(t)$ and $p_f = 1/n$, i.e., with a guarantee that the estimated PPR $\hat{\pi}(s,t)$ satisfies $|\hat{\pi}(s,t) - \pi(s,t)| \leq c \cdot \phi \cdot \pi(t)$ with $1 - 1/n$ probability. Then, it can distinguish whether $\pi(s,t) > (1+c) \cdot \phi \cdot \pi(t)$ or $\pi(s,t) < (1-c) \cdot \phi \cdot \pi(t)$. Next, we will show that we can provide a tighter bound if we want to identify whether s is an AHH of t. The main rationale behind the improved bound is that when the actual PPR is very large, then it is not necessary to provide a bound $\lambda = c \cdot \phi \cdot \pi(t)$, and can choose a looser bound to identify whether s is an AHH of t or not. Clearly, with a loose bound, the amount of sampling required by the Monte-Carlo approach will also decrease, which helps improve the query efficiency. Next, we show with an example the cases where we might be able to use a loose bound.

Example 3.1. Assume that ϕ, c are given and note that $c < 1$ according to Definition 2.4. Suppose for a given node s and node t, it holds that $\pi(s,t) > 100\phi \cdot \pi(t)$. Then, it suffices to set $\lambda = 98c \cdot \phi \cdot \pi(t)$. To explain, by setting $\lambda = 98c \cdot \phi \cdot \pi(t)$, we guarantee that with $1 - 1/n$ probability, the estimated PPR follows that:

$$\begin{aligned}\hat{\pi}(s,t) &> \pi(s,t) - 98c \cdot \phi \cdot \pi(t) > 100\phi \cdot \pi(t) - 98c \cdot \phi \cdot \pi(t) \\ &> 2 \cdot \phi \cdot \pi(t) > (1+c) \cdot \phi \cdot \pi(t).\end{aligned}$$

Therefore, even if we set λ much looser than $c \cdot \phi \cdot \pi(t)$, we can still correctly answer the pairwise AHH query. □

Inspired by the above observation, we present a new bound for the Monte-Carlo approach and the new results are summarised as shown in Proposition 3.2 .

Proposition 3.2. *Let X be a random variable in the range of $[0, r]$. Given $0 < \phi' \leq 1$ and $0 \leq c < 1$, to distinguish:*

- $E[X] > (1+c) \cdot \phi' \cdot r$,
- $E[X] < (1-c) \cdot \phi' \cdot r$.

with a failure probability at most p_f, it suffices to sample a number $O\left(\frac{\log(1/p_f)}{\phi'}\right)$ of random walks. □

With Proposition 3.2, it is not difficult to apply it on pairwise AHH queries to derive a new bound. In particular, given a source s and a target t, let X be a random variable such that if a random walk starting from s terminates at t then $X = 1$ and otherwise $X = 0$. Then, $r = 1$, and according to the definition of PPR, $E[X] = \pi(s,t)$. Besides, let $\phi' = \phi \cdot \pi(t)$ and $p_f = 1/n$. Based on Definition 2.6, the number of random walks required to answer the pairwise AHH query can be bounded by $O\left(\frac{\log n}{\phi \cdot \pi(t)}\right)$. Also recall that $\phi \cdot \pi(t) \leq 1$. Compared to the trivial bound $O\left(\frac{1}{(\phi \cdot \pi(t))^2} \cdot \log n\right)$ in Section 2.2, the new bound improves over the trivial bound by $\frac{1}{\phi \cdot \pi(t)}$.

3.2 Bidirectional Pairwise AHH

Next, we show how to use a bidirectional approach to solve pairwise AHH queries. The main idea of our proposed *BLOG-P (BLOG for Pairwise AHH)* is to combine the Monte-Carlo approach and the backward propagation algorithm to improve query efficiency. Given a target node t, it first starts a backward propagation from

t. The choice of r_{max} will be discussed later and recall that the following equation holds when backward propagation terminates:

$$\pi(s,t) = \boldsymbol{p}(s) + \sum_{v \in V} \pi(s,v) \cdot \boldsymbol{r}(v).$$

Besides, according to Algorithm 1, when the backward propagation terminates, $\boldsymbol{r}(v) < r_{max}$ for any node $v \in V$. Let X be a random variable such that it takes $\boldsymbol{r}(v)$ if a random walk starting from s terminates at v. According to the definition, $E[X] = \sum_{v \in V} \pi(s,v) \cdot \boldsymbol{r}(v)$. Also note that $X \leq r_{max}$. Define $Z = X/r_{max}$; note that $Z \in [0,1]$. After that, we can apply the following lemma to derive an estimate of Z, and hence an estimate of $\pi(s,t)$.

Lemma 3.3. *Let X be a random variable in the range of $[0,r]$, y be a positive real value. There is a value w such that $w \geq r + y$. Given $0 < \phi \leq 1, 0 \leq c < 1$, to distinguish:*

- $y + E[X] > (1+c) \cdot \phi \cdot w$,
- $y + E[X] < (1-c) \cdot \phi \cdot w$;

with a failure probability at most p_f, it suffices to have:

- *a number $k = O\left((\frac{\phi w - y}{\phi^2 w} + \frac{1}{\phi})\frac{r}{w} \log(1/p_f)\right)$ of random samples of X, if $\phi \cdot w - y \leq r, c \cdot \phi \cdot w \leq r$, and $y \leq \phi \cdot w$;*
- *$k = 0$ samples otherwise.*

Next, we show how to connect Lemma 3.3 to the pairwise AHH query. To explain, after the backward propagation, let $y = \boldsymbol{p}(s)$, $w = \pi(t)$, $r = r_{max}, p_f = 1/n$, and recall that $E[X] = \sum_{v \in V} \pi(s,v) \cdot \boldsymbol{r}(v)$ according to our definition of the random variable X. Then, based on Equation 2, $y + E[X] = \pi(s,t)$. Therefore, with Lemma 3.3, we can answer the pairwise AHH query, and the number of random walks can be bounded by $O\left((\frac{\phi\pi(t) - \boldsymbol{p}(s)}{\phi^2 \pi(t)} + \frac{1}{\phi})\frac{r_{max}}{\pi(t)} \log n\right)$. Note that when $\boldsymbol{p}(s) = 0$ and $r_{max} = 1$, the new bound will degrade to the same as the one presented in Section 3.1.

Algorithm details. The pseudo-code of our BLOG-P algorithm is as shown in Algorithm 2. Line 1 initializes r_{max} according to Equation 3. Algorithm 2 Lines 2-12 derives the estimated PPR $\hat{\pi}(s,t)$ for node t with respect to s. In Lines 2-8, the algorithm handles the case when r_{max} is no larger than 1, and in Lines 9-12 it handles the case when $r_{max} > 1$. Finally, Lines 13-16 identifies whether return s as an AHH of t or not. In particular, if $\hat{\pi}(s,t)$ is larger than $\phi \cdot \pi(t)$, then it returns true, and otherwise returns false.

Complexity Analysis. The cost of BLOG-P includes two parts: the backward propagation and the forward random walks. Given the backward threshold r_{max}, the backward propagation has a cost of $O\left(\frac{\pi(t) \cdot c_{push}}{r_{max}}\right)$. Also in the forward random walks, according to Proposition 3.3, the number of random walks can be bounded by $O\left((\frac{\phi\pi(t) - \boldsymbol{p}(s)}{\phi^2 \pi(t)} + \frac{1}{\phi})\frac{r_{max}}{\pi(t)} \log n\right)$. Note that $\frac{1}{\phi} \leq (\frac{\phi\pi(t) - \boldsymbol{p}(s)}{\phi^2 \pi(t)} + \frac{1}{\phi}) \leq \frac{2}{\phi}$ and the expected number of visited nodes in a random walk is $\frac{1}{\alpha}$, which is a constant. Hence, the expected cost of the forward random walks can be bounded by $O\left(\frac{r_{max}}{\phi \cdot \pi(t)} \log n\right)$. Combining these two phases, we have that the total cost of BLOG-P algorithm is:

$$O\left(\frac{\pi(t) \cdot c_{push}}{r_{max}} + \frac{r_{max}}{\phi \cdot \pi(t)} \log n\right).$$

When we set

$$r_{max} = \pi(t)\sqrt{\frac{\phi \cdot c_{push}}{\log n}}, \tag{3}$$

the above complexity is minimized. However, notice that in backward propagation, we need to set $r_{max} \leq 1$, and otherwise the cost of the backward propagation is zero and will not be $O\left(\frac{\pi(t) \cdot c_{push}}{r_{max}}\right)$. Therefore, it is possible that we do not achieve the minimized result since r_{max} derived according to Equation 3 may be larger than 1. Hence, we have the following two cases:

- **Case 1:** $\pi(t)\sqrt{\frac{\phi \cdot c_{push}}{\log n}} \leq 1$. We do backward propagation by setting r_{max} using Equation 3. Afterwards, we do $O\left(\sqrt{\frac{1}{\phi}} \cdot \sqrt{c_{push} \cdot \log n}\right)$ random walks from s. The time complexity of the bidirectional algorithm is $O\left(\sqrt{\frac{1}{\phi}} \cdot \sqrt{c_{push} \cdot \log n}\right)$;
- **Case 2:** $\pi(t)\sqrt{\frac{\phi \cdot c_{push}}{\log n}} > 1$. In this case, we simply do $O\left(\frac{\log n}{\phi \cdot \pi(t)}\right)$ random walks from s.

Combining these two cases, we have that *BLOG-P* has a complexity of $O\left(\min\{\sqrt{\frac{1}{\phi}} \cdot \sqrt{c_{push} \cdot \log n}, \frac{\log n}{\phi \cdot \pi(t)}\}\right)$. As we can see, this cost reduces the term $\frac{\log n}{\phi}$ of the Monte-Carlo approach to $\sqrt{\frac{\log n}{\phi}}$ but brings a new term $\sqrt{c_{push}}$. As we will see in our experiments, in practice, the impact of the reduced term $\sqrt{\frac{\log n}{\phi}}$ is more significant than the new term $\sqrt{c_{push}}$, and *BLOG-P* is three orders of magnitude faster than the Monte-Carlo approach with the tighter bound presented in Section 3.1.

4 DEALING WITH HIGH IN-DEGREE NODES

Recall that the cost of the backward propagation is $\frac{\pi(t) \cdot c_{push}}{\alpha r_{max}}$. In the worst case, c_{push} can be d^-_{max}, i.e., the maximum in-degree of nodes in the graph, which can be very large. For example, in Twitter social graph, some celebrities may have millions of followers, indicating a high in-degree for nodes representing celebrities. It is desirable to handle such high in-degree nodes separately during the backward propagation. For instance, we may pre-store backward propagation results for such nodes. Yet if a backward propagation starts from one of the followers of these high in-degree nodes, we still need to do pushback on such nodes and it still incurs high computational costs. In this section, we present techniques to handle such high in-degree nodes to improve the worst case query efficiency without compromising approximation guarantees.

4.1 Main idea

Usually, such high in-degree nodes, e.g., celebrities, only account for a small portion of the total nodes. Therefore, if we can avoid doing pushback operations on such a small portion of nodes without compromising the approximation guarantee, the worst case time complexity of our *BLOG-P* then can be improved. Let $(v_1, \cdots, v_k, \cdots, v_n)$ be the sequence of nodes sorted in descending order of their in-degrees. The idea is that for each node v_i which

Algorithm 2: BLOG-P Algorithm

Input: Graph G, source node s, target node t, probability α
Output: Whether t is an AHH of s or not

Calculate r_{max} according to Equation 3;
if $r_{max} \leq 1$ **then**
  Invoke Backward Propagation Algorithm (Algorithm 1);
  Let $k = 2(\frac{\phi\pi(t)-\boldsymbol{p}(s)}{(c\cdot\phi)^2\pi(t)} + \frac{1}{c\cdot\phi})\frac{r_{max}}{\pi(t)}\log n$;
  $\hat{\pi}(s,t) = \boldsymbol{p}(s)$;
  for *i from 1 to k* **do**
    Sample a random walk starting from s. Let v the the ending node of the random walk;
    $\hat{\pi}(s,t) += \frac{\boldsymbol{r}(v)}{k}$;
else
  Let $k = 2(\frac{1}{c^2} + \frac{1}{c}) \cdot \frac{\log n \cdot \pi(t)}{\phi}$;
  Sample k random walks, and let c be the number of random walks terminate at t;
  $\hat{\pi}(s,t) = c/k$;
if $\hat{\pi}(s,t) > \phi \cdot \pi(t)$ **then**
  return true;
else
  return false;

is one of the nodes with the top-k largest in-degrees, i.e., $i \leq k$, we pre-store the PPR $\pi(s, v_i)$ of v_i with respect to each source $s \in V$. Then, during the backward propagation, we only do pushback operations from nodes whose ranks are larger than k, and it terminates as soon as $\boldsymbol{r}(v)$ is smaller than r_{max} for any node v with rank larger than k. Clearly, with this strategy, we can avoid the expensive pushback operations from nodes with high in-degrees.

It remains to show that the modified version of the backward propagation, dubbed as *light-weighted backward propagation (LBP)*, can be used to derive the improved bound for BLOG-P. Recall that the backward propagation always guarantees that:

$$\pi(s,t) = \boldsymbol{p}(s) + \sum_{v \in V} \boldsymbol{r}(v) \cdot \pi(s,v).$$

The BLOG-P algorithm then generates a random variable $X \in [0, r_{max}]$ such that its expectation is equal to the term $\sum_{v\in V} \boldsymbol{r}(v) \cdot \pi(s,v)$ and hence can apply concentration inequalities to derive approximation for $\pi(s,t)$ so as to answer the pairwise AHH query. However, for the LBP, there are mainly two issues: *(i)* Could LBP terminate and provide the same time complexity as the original backward propagation algorithm? *(ii)* Could we still apply the bidirectional approach to derive an approximation for the PPR so as to answer the AHH query? For the second issue, on one hand, if we pre-store the PPRs for the high degree nodes, and stop doing pushback operations from the top-k nodes, the maximum $\boldsymbol{r}(v)$ value among all $v \in V$ may exceed r_{max} and can be as high as 1, in which case we cannot apply Equation 3 to derive the improved bound. On the other hand, it will be difficult to generate a random variable whose expectation is equal to the term $\sum_{v\in V} \boldsymbol{r}(v) \cdot \pi(s,v)$. Then, we may not be able to apply the concentration inequalities to derive approximation for $\pi(s,t)$ so as to answer the pairwise AHH query. Next, we present the analysis of the proposed LBP and resolve the two issues in the analysis.

4.2 Analysis of LBP

We first show that the LBP algorithm can terminate and finish with at most $\frac{\pi(t)}{\alpha r_{max}}$ pushback operations. The analysis mainly follows the ones presented in [1]. Observe that in the LBP algorithm, after each pushback operation, the L_1 norm of $\boldsymbol{p}$ increases by at least $r_{max} \cdot \alpha$. To explain, in each pushback operation, suppose this operation starts from v, then $\boldsymbol{r}(v) > r_{max}$ and α portion of $\boldsymbol{r}(v)$ is transferred to $\boldsymbol{p}(v)$. Also, note that $\boldsymbol{p}(v) \leq \pi(v,t)$. Hence, the L_1 norm of $\boldsymbol{p}$ will not exceed sums of the PPR of t with respect to each source $s \in V$, which is equal to $\pi(t)$ according to Definition 2.2, and the number of pushback operations can be bounded by $\frac{\pi(t)}{\alpha r_{max}}$.

Next, we explain how to extend the LBP to the bidirectional approach for the pairwise AHH query. Let H be the set of vertices that are among the nodes with top-k in-degrees. Rewrite the backward propagation invariant as follows.

$$\pi(s,t) = \boldsymbol{p}(s) + \sum_{v\in V\setminus H} \boldsymbol{r}(v) \cdot \pi(s,v) + \sum_{v\in H} \boldsymbol{r}(v) \cdot \pi(s,v),$$

Note that the term $\sum_{v\in H} \boldsymbol{r}(v) \cdot \pi(s,v)$ can be directly obtained with $O(|H|)$ time by accessing the pre-stored PPRs. Regarding the term $\sum_{v\in V\setminus H} \boldsymbol{r}(v) \cdot \pi(s,v)$, it is non-trivial to define a random variable such that its expectation is equal to $\sum_{v\in V\setminus H} \boldsymbol{r}(v) \cdot \pi(s,v)$, which is the key idea behind the bidirectional approach for the pairwise AHH query. However, we show that after a reformulation, we can still apply the solution as proposed in Section 3.2. In particular, we redefine:

$$\boldsymbol{r'}(v) = \begin{cases} \boldsymbol{r}(v), \text{ if } v \in V \setminus H, \\ 0, \text{ otherwise.} \end{cases} . \tag{4}$$

Then, $\sum_{v\in V\setminus H} \boldsymbol{r}(v) \cdot \pi(s,v) = \sum_{v\in V} \boldsymbol{r'}(v) \cdot \pi(s,v)$. Also, for any node $v \in V$, $\boldsymbol{r'}(v) \leq r_{max}$. Therefore, we can apply the similar analysis in Section 3.2 to obtain the similar result for BLOG-P Algorithm. In particular, we can further define a random variable whose expectation is $\sum_{v\in V} \boldsymbol{r'}(v) \cdot \pi(s,v)$ and apply the concentration inequalities to derive an approximate result for $\sum_{v\in V\setminus H} \boldsymbol{r}(v) \cdot \pi(s,v) = \sum_{v\in V} \boldsymbol{r'}(v) \cdot \pi(s,v)$ and hence the approximate result for $\pi(s,t)$. Then, the maximum in-degree of the node in $V \setminus H$ is d^-_{k+1}, and the cost of a pushback operation in the worst case can be reduced from d^-_{max} to d^-_{k+1}.

5 REVERSE APPROXIMATE HEAVY HITTER

In this section, we present the proposed solution for the reverse approximate heavy hitter query algorithm. Recall that the reverse AHH query also takes as input a source node s. However, different from the pairwise AHH query, it takes as input a target set T instead a single target node. Then, the query asks for the set H such that (i) if s is an absolute AHH of node $t \in T$, then node s must exist in H, and (ii) if s is not an AHH, then s must not exist in H.

Clearly, a naive solution to reverse AHH queries is to apply the BLOG-P algorithm for each target $t \in T$ with respect to s. However,

the complexity of this approach will be linear to the size of T, i.e.,

$$O\left(|T| \cdot \min\{\sqrt{\frac{1}{\phi}} \cdot \sqrt{c_{push} \cdot \log n}, \frac{\log n}{\phi \cdot \pi(t)}\}\right),$$

which is not a favourable choice.

However, it is worth noting that the Monte-Carlo approach proposed in Section 3.1, which is an inferior solution for the pairwise AHH query, actually works quite well for the reverse AHH query. To explain, for the Monte-Carlo approach, it derives an estimation of $\pi(s, v)$ for each $v \in V$. Let t_m be the node who has the minimum PageRank of all nodes in T. Then, if we can guarantee that for t_m, we answer an AHH query, so will be the case for all the other nodes in T. Hence, to answer reverse AHH queries, it suffices to sample $O\left(\frac{\log n}{\phi \cdot \pi(t_m)}\right)$ random walks. This approach, as we will see in the experiment, is still slow given a target set T with $|T| \ll |V|$.

In this section, we will first demonstrate in Section 5.1 our core idea, the *logarithmic bucketing* approach, which is a generalized technique that can be applied to both reverse AHH queries and multi-source reverse AHH queries. Then, in Sections 5.2 and 5.3, we present how our *BLOG-R (BLOG for reverse AHH)* handles the reverse AHH queries with our logarithmic bucketing approach. The pseudo-code of BLOG-R is shown in Appendix B.

5.1 Logarithmic bucketing

The main observation of our logarithmic bucketing approach is that in the naive solution, which applies the pairwise BLOG-P for s and each node $t \in T$, all the pairwise AHH queries will need to do the backward propagation algorithm from each target node, resulting in increased overhead. However, note that we can share the forward random walks and it is not necessary to repeat the forward random walks $|T|$ times. As a result, it is possible to tune up the forward random walk cost and tune down the backward propagation cost to achieve better query efficiency. However, the main challenge is that for each target node t, the cost to do the backward propagation from t will depend on $\pi(t)$. If we simply set the same r_{max} for all the nodes, it will be difficult to bound the $\pi(t)$ part since in the worst case their sum will be $O(n)$.

Our logarithmic bucketing approach is proposed to tackle this challenging issue. The high level idea of the logarithmic bucketing approach is to divide the nodes into buckets in a logarithmic manner as follows. Let π_{min} be the minimum PageRank of all nodes in T. Then, we divide the interval $[\pi_{min}, n]$ into $b = \lceil \log_2(n/\pi_{min}) \rceil$ disjoint intervals as $(n/2, n], (n/4, n/2], \cdots, [n/2^b, n/2^{b-1}]$. Let T_j be the set of nodes whose PageRank fall in the j-th interval. For nodes in T_j, we use the same r_{max} to do the backward propagation. With this approach, we can bound the cost of backward propagations within bucket T_j since the cost will differ by at most a factor of 2. By setting different r_{max} for different buckets, we can avoid the issue mentioned in the beginning, i.e., the total cost depends on the sum of $\pi(t)$ for $t \in T$, which is $O(n)$ in the worst case.

With the logarithmic bucketing approach, we further demonstrate that when taking into consideration of all the buckets, the final time complexity of our query algorithm *BLOG-R* for answering reverse AHH queries improves over the naive pairwise *BLOG-P* algorithm by a factor of $\sqrt{|T|}$. As we will see in our experiment, the logarithmic bucketing approach is so effective such that it is at least an order of magnitude faster than the pairwise approach.

Besides, our logarithmic bucketing idea can be further generalized to the multi-source reverse AHH queries. With the logarithmic bucketing approach, we can further reduce the dependency on the size of the source set $|S|$ to $\sqrt{S}$. The details will be explained in Section 6. In the following, we will explain our *BLOG-R* algorithm for answering the reverse AHH queries.

5.2 Reverse AHH within a Bucket

We first consider the subproblem that handles the case when all the target nodes are within the same bucket.

Problem 1. *Given a subset $T_s \subseteq T$ of vertices with the guarantee that $\pi(t) \in [w', 2w']$ for all $t \in T_s$. Given a source vertex s, we want to report a subset $R \subseteq T_s$ such that for each $t \in T_s$, it guarantees that:*

- *if s is an absolute AHH of t, then t must belong to R.*
- *if s ia a permissible AHH of t, then t may (not) belong to R.*
- *otherwise, t must not belong to R.*

with a success probability of $1 - 1/n$. □

To solve this subproblem, we still apply a bidirectional approach which gains a balance between the random walk cost and the backward propagation cost. Note that for each node $t \in T_s$, if we apply the *BLOG-P* algorithm, their PageRanks and the value of r_{max} are different, which makes the backward propagation cost differ from each other. Our strategy is to apply the same r_{max} for all backward propagations. Then, since $\pi(t) \in [w', 2w']$, the backward propagation cost can be bounded by $O\left(|T_s| \cdot \frac{w' \cdot c_{push}}{r_{max}}\right)$. In the meantime, to guarantee that for any node $t \in T_s$, we can determine whether t is an AHH of s or not, it suffices to sample $O\left(\frac{r_{max}}{\phi \cdot w'} \log n\right)$ random walks. Therefore, by adding the cost together, we have that the total cost to solve Problem 1 can be bounded by:

$$O\left(|T_s| \cdot \frac{w' \cdot c_{push}}{r_{max}} + \frac{r_{max}}{\phi \cdot w'} \log n\right). \tag{5}$$

It can be verified that when $r_{max} = w'\sqrt{\frac{|T_s| \cdot c_{push} \cdot \phi}{\log n}}$, the time complexity in Equation 5 is minimized, which is $O\left(\sqrt{\frac{1}{\phi}}\sqrt{|T_s| \cdot c_{push} \cdot \log n}\right)$. However, recall that r_{max} should not be larger than 1. When $r_{max} > 1$, we still apply the Monte-Carlo approach, which samples $O\left(\frac{\log n}{\phi \cdot w'}\right)$ random walks. Note that in this case, $\sqrt{\frac{|T_s| \cdot c_{push} \cdot \phi}{\log n}} > 1/w'$, which means $O\left(\frac{\log n}{\phi \cdot w'}\right)$ can be bounded by $O\left(\sqrt{\frac{1}{\phi}}\sqrt{|T_s| \cdot c_{push} \cdot \log n}\right)$. Therefore, Problem 1 can be solved with $O\left(\sqrt{\frac{1}{\phi}}\sqrt{|T_s| \cdot c_{push} \cdot \log n}\right)$ time.

5.3 Putting it together

Next, we explain how to combine the cost of all buckets and derive an improved bound. Let π_{min} be the minimum PageRank in T, it is worth noting that $\pi_{min} \geq \alpha$, since $\pi(t)$ is the sum of the PPR of each node with respect to t, and t itself has a PPR no smaller than α. Then, the total number b of buckets can be bounded by

$b = \log_2(n/\pi_{min}) \leq \log_2(n/\alpha) = O(\log n)$. Let T_j be the set of nodes in the j-th bucket. The total cost C can then be bounded by:

$$C = O(\sum_{j=1}^{b}(\sqrt{\frac{1}{\phi}}\sqrt{|T_j| \cdot c_{push} \cdot \log n})$$

$$= O(\sqrt{\frac{1}{\phi}} \cdot \sqrt{c_{push} \cdot \log n} \cdot \sum_{j=1}^{b}\sqrt{|T_j|})$$

Note that $(x_1 + \cdots + x_k)^2 \leq k \cdot (x_1^2 + \cdots + x_k^2)$, we have that:

$$C \leq O(\sqrt{\frac{1}{\phi}}\sqrt{c_{push} \cdot \log n} \cdot \sqrt{b} \cdot \sqrt{\sum_{j=1}^{b}|T_j|})$$

$$= O(\sqrt{\frac{1}{\phi}}\sqrt{|T| \cdot c_{push}} \cdot \log n).$$

Here, it still needs further discussion to derive the final complexity for the reverse AHH problem.

- **Case 1.** $|T| > \frac{1}{\phi \cdot c_{push}}$. In this case, we then simply use the Monte-Carlo approach as proposed in Section 3.1, which has a complexity of $O\left(\frac{\log n}{\pi_{min} \cdot \phi}\right)$, where $\pi_{min} = \min_{t \in T} \pi(t)$.
- **Case 2.** $|T| \leq \frac{1}{\phi \cdot c_{push}}$. In this case, we use the logarithmic bucketing approach, which has a complexity of $O(\sqrt{\frac{1}{\phi}}\sqrt{|T| \cdot c_{push} \cdot \log n})$.

Combining the two cases, we have the final time complexity of *BLOG-R* for the reverse AHH query, which is $O\left(\min\{\frac{\log n}{\phi \cdot \pi_{min}}, \sqrt{\frac{1}{\phi}}\sqrt{|T| \cdot c_{push}} \cdot \log n\}\right)$, improving over the solution using *BLOG-P* algorithm by $\sqrt{|T|}$ when $|T| \leq \frac{1}{\phi \cdot c_{push}}$.

6 MULTI-SOURCE REVERSE AHH

In this section, we show how to apply our logarithmic bucketing idea to answer multi-source reverse AHH queries, in which case, the source is a set S instead of a single node compared to reverse AHH queries. The query outputs the reverse AHH query with the target set T for each source $s \in S$. A naive solution is to apply the BLOG-R algorithm $|S|$ times and the time complexity is $O\left(|S| \min\{\frac{\log n}{\phi \cdot \pi_{min}}, \sqrt{\frac{1}{\phi}}\sqrt{|T| \cdot c_{push}} \cdot \log n\}\right)$ which depends linear to the size of S. As we will show, by carefully balancing forward and backward cost with our logarithmic bucketing approach, we can improve over the naive solution by up to $\sqrt{|S|}$. The proposed algorithm is denoted as *BLOG-M (BLOG for multi-source reverse AHH)*. The pseudo-code of BLOG-M is shown in Appendix B.

Rationale. The main observation is that in BLOG-R, the forward cost $O(\frac{r_{max}}{\phi \cdot w'} \log n)$ does not depend on the source node, and is the same for any given source node. Therefore, given a source set S, the forward cost can be bounded by $O(|S|\frac{r_{max}}{\phi \cdot w'} \log n)$. With this, we can adjust r_{max} so as to balance the forward and backward cost within a bucket. To consider the forward and backward cost within a bucket, it is still suffices to consider the following subproblem.

Problem 2. *Given a subset T_s of vertices with the guarantee that $\pi(t) \in [w', 2w']$ for all $t \in T_s$. Given every source vertex $s \in S$, we want to report a subset $R_s \in T_s$, such that for each $t \in T_s$, it guarantees that:*

- *if s is an absolute AHH of t, then t must belong to R_s.*
- *if s is a permissible AHH of t, then t may (not) belong to R_s.*
- *otherwise, t must not belong to R_s.*

with a success probability of $1 - 1/n$.

For the above problem, the forward random walks will have a cost of $O\left(\frac{r_{max} \cdot |S|}{\phi \cdot w'} \cdot \log n\right)$, while the backward propagation from each target node has a cost of $O\left(\frac{w' \cdot c_{push} \cdot |T_s|}{r_{max}}\right)$, and in total, the subproblem can be solved with a time complexity of :

$$O\left(\frac{r_{max} \cdot |S|}{\phi \cdot w'} \cdot \log n + \cdot \frac{w' \cdot c_{push} \cdot |T_s|}{r_{max}}\right) \quad (6)$$

By setting $r_{max} = w'\sqrt{\frac{\phi \cdot c_{push}}{\log n}} \cdot \sqrt{\frac{|T_s|}{|S|}}$, Equation 6 obtains the minimized complexity $O\left(\sqrt{\frac{1}{\phi}}\sqrt{|S| \cdot |T_s| \cdot c_{push} \cdot \log n}\right)$. Recall that it requires r_{max} to be no larger than 1 to run the backward propagation algorithm. When $r_{max} > 1$, i.e., $w'\sqrt{\frac{c_{push}}{\lambda \cdot \log n}}\sqrt{\frac{|T_s|}{|S|}} > 1$, we can simply run the Monte-Carlo approach from each source node in S, which has a cost of $O\left(\frac{\log n \cdot |S|}{\phi \cdot w'} \cdot\right)$. It is easy to verify that when $r_{max} > 1$, $O\left(\frac{|S|}{\phi \cdot w'} \cdot \log n\right)$ can be bounded by $O\left(\sqrt{\frac{1}{\phi}}\sqrt{|S| \cdot |T_s| \cdot c_{push} \cdot \log n}\right)$. Therefore, the cost to solve Problem 2 can be bounded by $O\left(\sqrt{\frac{1}{\phi}}\sqrt{|S| \cdot |T_s| \cdot c_{push} \cdot \log n}\right)$.

Comparing the cost of Problem 2 and Problem 1, Problem 2 brings an additional $\sqrt{|S|}$ term. Nevertheless, to consider the cost of all buckets, we can still apply the similar approach as mentioned in Section 5.3 since $\sqrt{|S|}$ is a constant for a given multi-source AHH query. Recall that the total cost of of reverse AHH query can be bounded by $O(\sqrt{\frac{1}{\phi}}\sqrt{|T| \cdot c_{push}} \log n)$. Since the multi-source AHH query only brings an additional $\sqrt{|S|}$ term, the cost of multi-source AHH query can then be bounded by $O(\sqrt{\frac{1}{\phi}}\sqrt{|S| \cdot |T| \cdot c_{push}} \log n)$.

Define $\pi_{min} = \min_{t \in T} \pi(t)$, then, we have two cases:

- $|T| \cdot c_{push} > \frac{|S|}{\phi}$, in this case, we can simply apply the Monte-Carlo approach $|S|$ times, which has a cost of $O(\frac{|S| \cdot \log n}{\phi \cdot \pi_{min}})$.
- $|T| \cdot c_{push} \leq \frac{|S|}{\phi}$, in this case, we apply the logarithmic bucketing approach, and derive the bound $O(\sqrt{\frac{1}{\phi}}\sqrt{|S| \cdot |T| \cdot c_{push}} \cdot \log n)$.

The total cost of our *BLOG-M* for answering the multi-source reverse AHH query is $O(\min\{\frac{|S| \cdot \log n}{\phi \pi_{min}}, \sqrt{\frac{1}{\phi}}\sqrt{|S| \cdot |T| \cdot c_{push}} \cdot \log n\})$, improving over the naive solution, which invokes BLOG-R $|S|$ times, by a factor of $\sqrt{|S|}$. The pseudo-code and description of the code can be found in Appendix.

7 RELATED WORK

PageRank and personalized PageRank are two fundamental metrics proposed by Page et al. [26] to measure the importance of a node in the graph. For the interest of space, we refer readers to a

detailed survey on PageRank [7], and focus on existing works on personalized PageRank. Existing solutions for personalized PageRank can be roughly divided into three categories: matrix-based [8, 9, 12, 18, 19, 24, 32], local-update based [2, 7, 13, 14, 18, 25, 31], and Monte-Carlo based approaches [5, 11, 22, 23, 27, 29].

The *matrix-based* approach, mainly relies on the matrix based definition of personalized PageRank as follows:

$$\boldsymbol{\pi_s} = \alpha \cdot \boldsymbol{e_s} + (1-\alpha) \cdot A^T D^{-1} \boldsymbol{\pi_s}, \tag{7}$$

where $\boldsymbol{\pi_s}$ is the PPR vector with the i-th entry storing $\pi(s, v_i)$, α is the decay factor defined in Section 2, $\boldsymbol{e_s}$ is a unit vector with a single nonzero entry at s, A is the adjacency matrix, and D is a diagonal matrix where the i-th entry equals the out-degree of node v_i. The main idea is to make an initial guess on $\boldsymbol{\pi_s}$ and repeatedly use Equation 7 to get new estimations of $\boldsymbol{\pi_s}$. The research work in this category [12, 19, 20, 24, 28, 32] mainly exploit the characteristics of graph structures, and decompose the adjacent matrix into sub-matrices so as to improve the matrix multiplication efficiency. The state-of-the-art solution in this category is *BePI* [19], which exploits the hub-and-spike structure for node reordering and block elimination. They further sparsify a matrix term used in the PPR calculation and apply a pre-conditioner to the iterative method to improve the query efficiency. Their approach is shown to scale to billion node graphs. Nevertheless, such methods are not applicable to our problem since they will compute the PPRs for all the nodes with respect to a source, resulting in high computational costs.

Another category is the local-update based approaches. The representatives are the *backward propagation* algorithm [1, 18] and the *forward push* algorithm [2]. There exist a lot of follow-up research works based on these two algorithms. The first line of research work [7, 13] mainly focus on how to improve the query efficiency by indexing or using distributed computing. The other line of research work [14, 25, 31] focus on applying these two algorithms on dynamic graphs. They assume that all the forward push or the backward propagation results are pre-computed and study how to update the pre-stored results so as to reflect the graph changes. However, only *backward propagation* algorithm can solve our problem, and as shown in our experiment, our *BLOG* framework is more efficient than backward propagation in all three queries.

Finally, there exist a plethora of research work that apply the Monte-Carlo approach to solve the personalized PageRank problem. The solutions in this category mainly rely on the random walk based definition of personalized PageRank. The first line of research work [4, 11, 27] study on how to improve the query efficiency using purely random walk based solution through indexing or distributed computing. Another line of research work [22, 23, 29, 30] study on how to combine random walks with the locate update algorithms so as to improve the query efficiency. BiPPR [22] is the state of the art for answering pairwise PPR queries without indexing techniques. The main idea is to combine the random walks and the backward propagation algorithm. However, simply applying *BiPPR* with absolute error guarantees will result in inferior performance according to our analysis in Section 3.1. Besides, we present new techniques to handle high in-degree nodes so as to improve the worst case query efficiency. Moreover, simply applying BiPPR to reverse AHH queries or the multi-source reverse AHH queries will be very inefficient, and we demonstrate techniques to tackle such issues. *HubPPR* subsumes *BiPPR*, and is an indexing based solution to improve the efficiency of the pairwise PPR query. The index structure also include pre-storing backward propagation results from some hub nodes. However, the solution cannot be similarly applied to our problem since in our problem ϕ can be given at the query time, while in HubPPR the pre-stored results depend on the input relative error guarantee. Most recently, Wang et al. [30] propose to combine the forward push and random walks to improve the query efficiency on top-k PPR queries, which returns top-k nodes with highest PPRs with respect to a source s. The solution is still ineffective to our problem since the solution still calculates the PPRs for all nodes with respect to the source s.

Name	n	m	Type	Max. in-degree
DBLP	613.6K	2.0M	undirected	588
Youtube	3.2M	8.9M	undirected	91751
Pokec	1.6M	30.6M	directed	13,733
Flickr	2.3M	33.1M	directed	21,001
LiveJournal	4.8M	69.0M	directed	13,905
Orkut	3.1M	117.2M	undirected	33,313
Twitter	41.7M	1.5B	directed	770,155

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

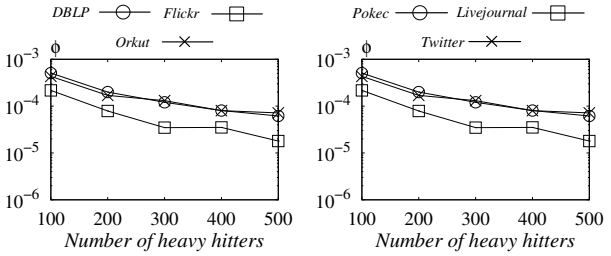
8 EXPERIMENTS

In this section, we experimentally evaluate our proposed BLOG framework for the three types of heavy hitter queries against the states of the art. All the algorithms are implemented with C++ compiled with O3 optimization. All the experiments are conducted on a Linux machine with an Intel 2.9GHz CPU and 400GB memory.

8.1 Experimental setting

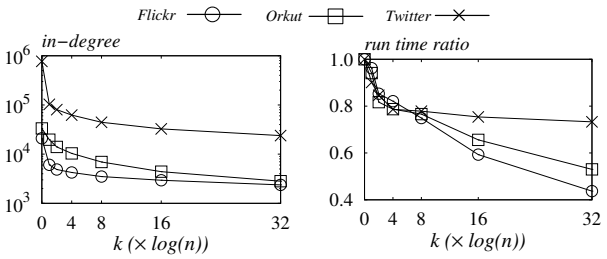
Datasets and query sets. We use 6 real datasets: *DBLP*, *Pokec*, *Flickr*, *Livejournal*, *Orkut*, and *Twitter* which are datasets widely used in existing research work [19, 22, 23, 28–30] on personalized PageRank. Table 2 summarizes the statistics of the datasets. To evaluate the pairwise AHH query, we randomly generate 1000 queries with source node and target node uniformly chosen as the query set. For the reverse AHH query, we first randomly generate 50 sources, and then for each source node, we vary the size of the target set T to examine the impact of $|T|$ to the performance of each algorithm. We vary $|T|$ with $\{100, 200, 400, 800, 1600, 3200, 6400\}$ for each dataset. The nodes in the target set T is also uniformly chosen from V. For the multi-source reverse AHH query, we have two sets of experiments. In the first set of experiments, we fix $|T|$ to 800 and vary the source set size with $\{100, 200, 400, 800, 1600, 3200, 6400\}$. In the second set of experiments, we fix $|S|$ to 800 and vary the target set size with $\{100, 200, 400, 800, 1600, 3200, 6400\}$. We generate 50 queries for each set of the experiments to evaluate the performance of the multi-source reverse AHH queries. We report the average running time as the query performance of each method.

Methods. For all the three queries, we include the *Monte-Carlo* approach with the tighter bound (Ref. Section 3.1), the *backward propagation* algorithm (Ref. Section 2.2) as the baseline solutions.



(a) (b)

Figure 1: Number of heavy hitters with respect to ϕ.



(a) (b)

Figure 2: Improvement to worst case query efficiency.

Besides, we also include BiPPR, the state-of-the-art method to estimate pairwise PPR scores as our competitor. Notice that BiPPR only provides relative guarantee on the estimated PPR. We derive the solution (Ref. Appendix C for the details) so that it can provide absolute guarantee to the PPRs and therefore can be used to answer our heavy hitter problem. For reverse and multi-source AHH queries, we apply our logarithmic bucketing approach to BiPPR such that the forward cost and backward cost is balanced. For the pairwise AHH query, we mainly evaluate the query efficiency of our proposed *BLOG-P* against the three baseline solutions. For the reverse AHH query, we compare our *BLOG-R* against the three baseline solutions. Besides, we also include our pairwise algorithm *BLOG-P* as a baseline, which simply applies the *BLOG-P* algorithm multiple times for each target t in the target set of the reverse AHH query. For the multi-source reverse AHH query, we further compare our proposed *BLOG-M* against the three baseline solutions. Besides, we also include our *BLOG-R* as a baseline, which applies the *BLOG-R* algorithm multiple times for each source in the source set of the multi-source reverse AHH query.

Parameter settings. Recall that in random walks, there is a decay factor α, which is usually set in the range $[0.15, 0.2]$. Following previous work [22, 23, 23, 29], we set $\alpha = 0.2$. Besides, in the backward propagation, we have the c_{push} parameter as the average cost of a pushback operation. To estimate the value, we randomly select 10000 target node and do the backward propagation algorithm from each target node. Then, we use the average cost of a pushback operation as c_{push}. Also, in the heavy hitter query, we have an input ϕ as the threshold to distinguish nodes who contribute more than ϕ portion of the PageRank of a target node. To have a understanding on the choice of ϕ, we have the following experiment. We randomly select 1000 nodes and calculate their maximum ϕ to return $\{100, 200, 300, 400, 500\}$ heavy hitters, and report the average ϕ. As shown in Figure 1, the x-axis shows the number of heavy hitters, and the y-axis shows the highest ϕ to return the required number of heavy hitters. Note that y-axis is in log-scale. Interestingly, to obtain a sufficient number of heavy hitters, e.g., 500 (this number is also used in the Who-To-Follow service in Twitter), ϕ needs to be set as low as 1×10^{-5}. In our following experiment, we set ϕ to be 1×10^{-5} according to this observation. Also, recall that the AHH definition requires an input of constant c. We set $c = 0.1$, in which case it guarantees that we will not admit a node s as an AHH of t if $\pi(s, t)$ is below 0.9ϕ portion of $\pi(t)$.

	MC	*Backward*	*BiPPR*	*BLOG-P*
DBLP	56.1K	158.1	4.1K	5.8
Pokec	86.9K	2.02K	11.7K	9.8
Flickr	83.2K	524.8	11.9K	7.7
LiveJournal	98.7K	722.1	18.2K	14.3
Orkut	158.2K	9.1K	25.8K	21.4
Twitter	424.8K	1.2K	91.4K	40.2

Table 3: Pairwise AHH query performance (ms). ($K = 10^3$)

8.2 Pairwise AHH queries

For the pairwise AHH queries, we have two sets of experiment. In the first set of experiment, we examine the average performance of our *BLOG-P* against existing solutions. Then, in the second set of experiment, we examine the effectiveness of our LBP algorithm to deal with high in-degree nodes on worst case query efficiency.

Table 3 reports the query efficiency of our *BLOG-P*, the Monte-Carlo approach (dubbed as MC in Table 3), the backward propagation algorithm (dubbed as Backward in Table 3) and BiPPR. As we can observe, our *BLOG-P* is always the most efficient algorithm and outperforms the competitors by orders of magnitude. In particular, our *BLOG-P* is at least three orders of magnitude faster than the Monte-Carlo approach on all the tested datasets, an order of magnitude faster than the backward propagation algorithm, and two order of magnitude faster than BiPPR. Despite the fact that BiPPR is the state-of-the-art approach to estimate PPRs, it achieves inferior performance on heavy hitter queries since it discards the fact that in heavy hitter queries, we only need to check whether the PPR is larger or smaller than the pre-defined threshold. In contrast, our *BLOG-P* captures this feature and significantly helps improve the query efficiency for the pairwise AHH queries. This demonstrates the effectiveness of the our *BLOG-P* algorithm.

Next, we further examine the worst case query efficiency of our *BLOG-P* algorithm. We evaluate the performance by pre-storing the PPR for the top $\{\lceil \log_2 n \rceil, 2\lceil \log_2 n \rceil, 4\lceil \log_2 n \rceil, 8\lceil \log_2 n \rceil, 16\lceil \log_2 n \rceil, 32\lceil \log_2 n \rceil\}$ nodes and use the techniques in Section 4 to handle the queries. Figure 2(a) reports the decrease of the worst case c_{push} on three representative datasets: Flickr, Orkut, and Twitter. As we can see, when we use the technique in Section 4, the worst case c_{push} can be reduced by an order of magnitude on all the tested datasets when we pre-store the PPRs for the top $32\lceil \log_2 n \rceil$ nodes. In case where the query efficiency is the most important concern, e.g., for web services need real-time responses,

our proposed solution can be applied so as to improve user experiences. We further empirically evaluate the impact of our methods to the improvement of our solution to the worst case query efficiency. However, it is rather difficult to design an effective approach to examine the improvement of our solution to the worst case query efficiency since it is not clear which node will result in the worst case. Therefore, we design the following qualitative analysis. We randomly generate 100, 000 queries, and report the average query time of the slowest 1%. Figure 2(b) shows the relative performance using our solution with the increase of the number of pre-stored nodes. As we can see, the average query time reduces with the increase of the pre-stored nodes, and with our technique, *BLOG-P* can save up to 60% query time when we pre-store the PPRs for the top $32\lceil \log_2 n \rceil$ nodes. This qualitatively evaluation further confirms the effectiveness of our solution on improving the worst case query efficiency. In Appendix D, we also include the experimental evaluation of the preprocessing costs of our LBP.

In summary, the experimental result suggests that our *BLOG-P* is a preferred choice on the pairwise AHH queries. Besides, our technique to handle the high in-degree nodes is also effective to improve the worst case query efficiency.

8.3 Reverse AHH queries

In the second set of experiments, we evaluate the query performance of all the methods for reverse AHH queries. Figures 3 reports the average query time of each method on four representative datasets: Pokec, Flickr, Orkut, and Twitter. In this set of experiments, we mainly examine the effectiveness of our proposed logarithmic bucketing approach with varying target set sizes.

From Figures 3(a)-(d). The main observation is that our *BLOG-R*, designed for answering reverse AHH queries, achieves the best performance on all the datasets. For instance, on Pokec, our *BLOG-R* is two orders of magnitude faster than the Monte-Carlo approach when $|T| = 100$, and is still 5x faster than the Monte-Carlo approach when the size of the target set increases to 6400. Compared to the backward propagation algorithm, our *BLOG-R* is two orders of magnitude faster. Moreover, *BLOG-R* is at least 5x faster than *BLOG-P* in most datasets when the target set size reaches 6400. This is because our *BLOG-R* applies the logarithmic bucketing approach, and adjusts the cost of the backward propagation by taking into account the number of nodes within a bucket to provide better query efficiency. Our *BLOG-P*, which is designed for the pairwise AHH query, is also very competitive and outperforms Monte-Carlo, backward propagation, and BiPPR in all cases.

In summary, the experimental result suggests that our logarithmic bucketing approach is very effective, and *BLOG-R* is the preferred choice to answer the reverse AHH query.

8.4 Multi-source reverse AHH queries

Finally, we evaluate our *BLOG-M* algorithm against other competitors for multi-source reverse AHH queries. We still report the results on four representative datasets: Pokec, Flickr, Orkut, and Twitter. We have two sets of experiments. The first set of experiments varies the size of the target size from 100 to 6400 with a fixed size of source set $|S| = 800$. The results are as shown in Figure 4. In the second set of experiments, the size of the target set is fixed to 800, and the size of the source set varies from 100 to 6400. Figure 5 reports the query efficiency of all the methods on the second set of experiments. Note that y-axis is in log-scale. We note that on some datasets, the performance of the Monte-Carlo approach is not reported since it cannot finish one query within 12 hours.

Firstly, observe from Figures 4(a)-(d) that our *BLOG-M* is the most efficient algorithm among all the competitors no matter how we vary the size of the target set. In particular, our *BLOG-M* is up to three orders of magnitude faster than backward propagation algorithm, four orders of magnitude than BiPPR, and five orders of magnitude than Monte-Carlo algorithm. Our *BLOG-M* is also an order of magnitude faster than the naive *BLOG-R* since *BLOG-M* applies the logarithmic bucketing approach and considers both the source set size and the target set size to balance the costs.

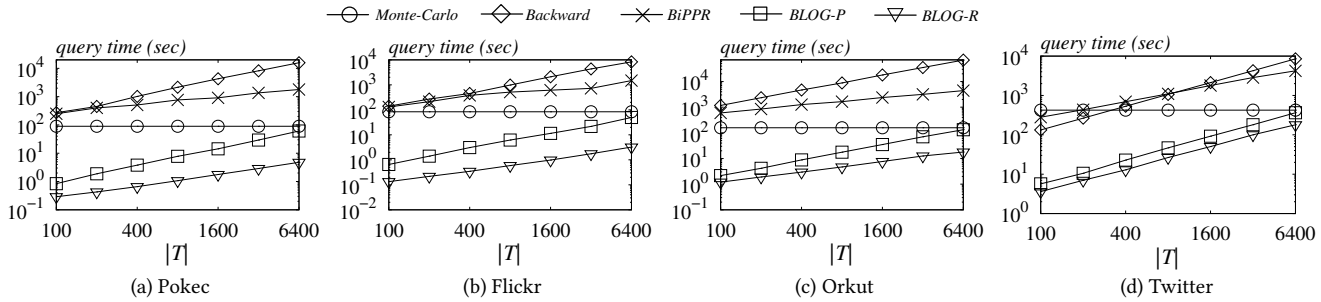
When we vary the size of the source set, as shown in Figures 5(a)-(d), our BLOG-M is still the most efficient algorithm and is orders of magnitude faster than other competitors. In particular, our BLOG-M is four orders of magnitude faster than the Monte-Carlo approach, three orders of magnitude faster than BiPPR, and up to two order of magnitude faster than the backward propagation algorithm. Besides, BLOG-M is up to two order of magnitude faster than BLOG-R, since BLOG-M, which further demonstrates the effectiveness of the proposed logarithmic bucketing approach.

In summary, the experimental results suggest that our *BLOG-M* algorithm is effective for multi-source reverse AHH queries, and is the preferred choice for multi-source reverse AHH queries.

8.5 Heavy Hitter for friend recommendation

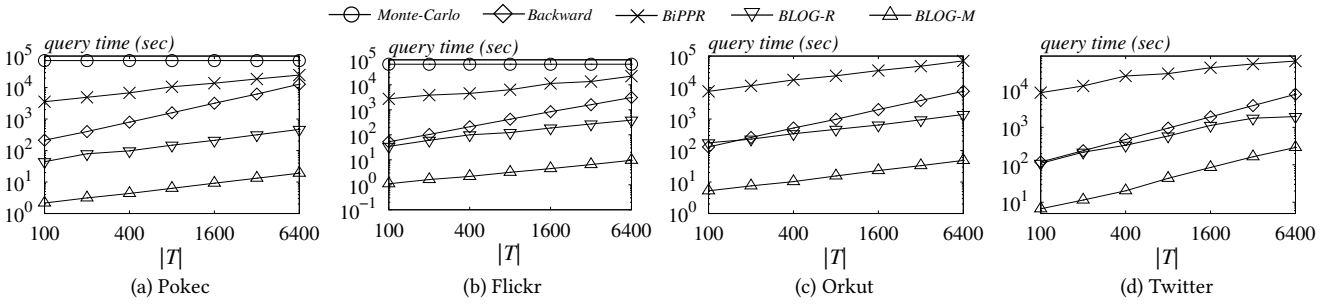
Finally, we examine the effectiveness of heavy hitters in personalized PageRank for friend recommendations. Our experimental settings are as follows: we test on the *Flickr* dataset, which includes additional timestamp information on each edge. Following previous work [21], we use the edges before a timestamp t to predict the existence of a certain edge after timestamp t. In our experiment, we set t such that 95% of the edges exist before t. Then, we consider the set U of users with at least 5 new edges after t, and randomly select 1000 users from U. Then, the evaluated method recommends a set R of 500 users to each user u in U, and if there indeed exists an edge from u to $v \in R$, we say that the prediction hits a recommendation. The intuition is that the larger the number of hit prediction is, the more accurate the recommendation algorithm is. We use the hit prediction as an indicator of the effectiveness of a recommendation algorithm. We report the number of hit prediction, and also the hit ratio, i.e., the number of hit predictions over the total number of predictions of each method, in our experiment.

For personalized PageRank, we select the top-500 users with the highest PPR with respect to a user u and recommend these 500 users to user u. As we mentioned in Section 1, PPR only considers one direction of importance. For heavy hitters, we consider two directions of importance. In particular, we first sort the nodes in decreasing order of their PPR values with respect to u, and this order indicates the importance of each user from the view point of u. Then, we calculate the ratio $r(v)$ of PPR $\pi(u, v)$ to the global PageRank $\pi(v)$ for each node v, and then sort the nodes in descending order of the ratio. This order indicates the importance of u from the viewpoint of each user. Then, we simultaneously scan these



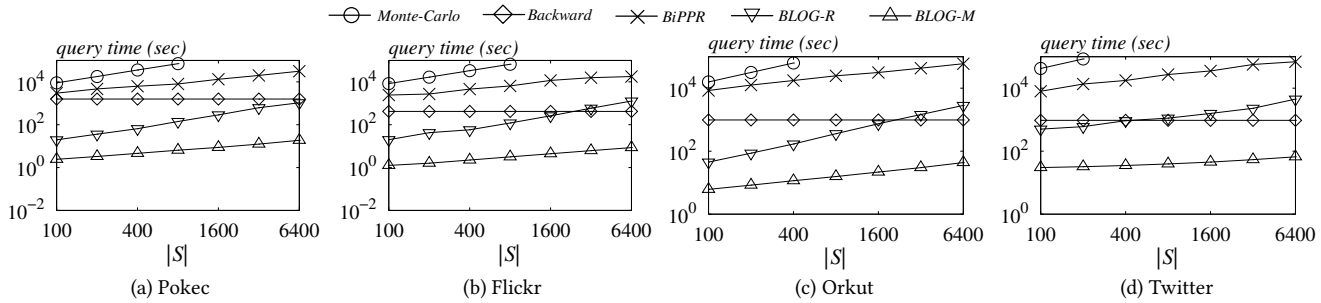
(a) Pokec (b) Flickr (c) Orkut (d) Twitter

Figure 3: Reverse AHH query efficiency.



(a) Pokec (b) Flickr (c) Orkut (d) Twitter

Figure 4: Multi-source reverse AHH query efficiency: varying $|T|$.



(a) Pokec (b) Flickr (c) Orkut (d) Twitter

Figure 5: Multi-source reverse AHH query efficiency: varying $|S|$.

	# of hit predictions		hit ratio	
# of new edges	PPR	HH	PPR	HH
54615	8396	8981	1.68%	1.80%

Table 4: Recommendation effectiveness on Flickr dataset.

two lists in order and record the number of common nodes in these two lists. The scan stops as soon as there are 500 common nodes that appear in the scanned part of both lists. Then, these 500 users are recommended to u.

The results are shown in Table 4. For the 1000 selected users, there are in total 54615 new edges added. By using heavy hitters (denoted as HH in Table 4), the number of hit prediction is 8981, improving over PPR by around 7%. As we can see, the hit ratio of both methods are less than 2%, which is consistently with the findings in [15]. As suggested in our experiment, by considering two directions of importance, heavy hitters can help improve the recommendation accuracy over PPR. This demonstrates the effectiveness of heavy hitters for friend recommendations.

9 CONCLUSION

This paper presents *BLOG*, an efficient framework for answering three types of approximate heavy hitter (AHH) queries. For pairwise AHH queries, we present how to combine the Monte-Carlo approach and the backward propagation tailored for pairwise AHH queries; we show how to handle high in-degree nodes to improve worst case query efficiency but still provide approximation guarantees. We further propose the logarithmic bucketing approach, which groups target nodes with similar PageRanks together and handle them in a batch to improve query efficiency of both reverse AHH queries and multi-source reverse AHH queries. Extensive experiments show that our solutions are orders of magnitude faster than alternatives under the same approximation guarantee.

10 ACKNOWLEDGMENTS

The research of Yufei Tao was partially supported by a direct grant (Project Number: 4055079) from CUHK and by a Faculty Research Award from Google.

REFERENCES

[1] Reid Andersen, Christian Borgs, Jennifer T. Chayes, John E. Hopcroft, Vahab S. Mirrokni, and Shang-Hua Teng. 2007. Local Computation of PageRank Contributions. In *WAW*. 150–165.

[2] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. 475–486.

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

[4] Lars Backstrom and Jure Leskovec. 2011. Supervised random walks: predicting and recommending links in social networks. In *WSDM*. 635–644.

[5] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. 2011. Fast personalized PageRank on MapReduce. In *SIGMOD*. 973–984.

[6] András A. Benczúr, Károly Csalogány, Tamás Sarlós, and Máté Uher. 2005. SpamRank – Fully Automatic Link Spam Detection. In *AIRWeb*. 25–38.

[7] Pavel Berkhin. 2005. Survey: A Survey on PageRank Computing. *Internet Mathematics* 2, 1 (2005), 73–120.

[8] Pavel Berkhin. 2006. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics* 3, 1 (2006), 41–62.

[9] Soumen Chakrabarti. 2007. Dynamic personalized pagerank in entity-relation graphs. In *WWW*. 571–580.

[10] Fan R. K. Chung and Lincoln Lu. 2006. Survey: Concentration Inequalities and Martingale Inequalities: A Survey. *Internet Mathematics* 3, 1 (2006), 79–127.

[11] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358.

[12] Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka. 2012. Efficient personalized pagerank with accuracy assurance. In *KDD*. 15–23.

[13] Tao Guo, Xin Cao, Gao Cong, Jiaheng Lu, and Xuemin Lin. 2017. Distributed Algorithms on Exact Personalized PageRank. In *SIGMOD*. 479–494.

[14] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel Personalized Pagerank on Dynamic Graphs. *PVLDB* 11, 1 (2017), 93–106.

[15] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. Wtf: The who to follow service at twitter. In *WWW*. 505–514.

[16] Zoltán Gyöngyi, Pavel Berkhin, Hector Garcia-Molina, and Jan O. Pedersen. 2006. Link Spam Detection Based on Mass Estimation. In *VLDB*. 439–450.

[17] Zoltan Gyongyi, Hector Garcia-Molina, and Jan Pedersen. 2006. *Web content categorization using link information.* Technical Report. Stanford.

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

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

[20] Jinhong Jung, Kijung Shin, Lee Sael, and U. Kang. 2016. Random Walk with Restart on Large Graphs Using Block Elimination. *ACM Trans. Database Syst.* 41, 2 (2016), 12:1–12:43.

[21] David Liben-Nowell and Jon M. Kleinberg. 2007. The link-prediction problem for social networks. *JASIST* 58, 7 (2007), 1019–1031.

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

[23] Peter Lofgren, Siddhartha Banerjee, Ashish Goel, and C Seshadhri. 2014. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *KDD*. 1436–1445.

[24] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. 2014. Computing personalized PageRank quickly by exploiting graph structures. *PVLDB* 7, 12 (2014), 1023–1034.

[25] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. 2015. Efficient PageRank Tracking in Evolving Networks. In *SIGKDD 2015*. 875–884.

[26] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).

[27] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. 2013. Fast Distributed PageRank Computation. In *ICDCN*. 11–26.

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

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

[30] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. In *SIGKDD*. 505–514.

[31] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate Personalized PageRank on Dynamic Graphs. In *KDD*. 1315–1324.

[32] Fanwei Zhu, Yuan Fang, Kevin Chen-Chuan Chang, and Jing Ying. 2013. Incremental and Accuracy-Aware Personalized PageRank through Scheduled Approximation. *PVLDB* 6, 6 (2013), 481–492.

APPENDIX

A THEOREMS AND PROOFS

Theorem A.1 (Chernoff Bound[10]). *Let $\hat{Y}$ be the average of ω i.i.d random variables sampled from a distribution on $[0,1]$, with a mean $\mathbb{E}[Y]$, for any ϵ,*

$$Pr[\hat{Y} - \mathbb{E}[Y] \geq \epsilon \cdot \mathbb{E}[Y]] \leq \exp\left(-\frac{\epsilon^2}{2+\epsilon}\omega \cdot \mathbb{E}[Y]\right),$$

$$Pr[\hat{Y} - \mathbb{E}[Y] \leq -\epsilon \cdot \mathbb{E}[Y] \leq \exp\left(-\frac{\epsilon^2}{2}\omega \cdot \mathbb{E}[Y]\right).$$

Proof of Lemma 2.9. Let Y be the random variable depending a random walk R from s, and its value is 1 if it terminates at t and otherwise 0. According to Definition 2.1, we know that $\mathbb{E}[Y] = \pi(s,t)$, and is in the range $[0,1]$. Let $\epsilon = \frac{\lambda}{\pi(s,t)}$. According to Theorem A.1, we have that:

$$Pr[|\hat{\pi(s,t)} - \pi(s,t)| \geq \epsilon \cdot \mathbb{E}[Y]] \leq \exp\left(-\frac{\epsilon^2}{2+\epsilon}\omega \cdot \mathbb{E}[Y]\right)$$

$$\Leftrightarrow Pr[|\hat{\pi(s,t)} - \pi(s,t)| \geq \lambda] \leq \exp\left(-\frac{\lambda^2}{2\pi(s,t)+\lambda}\omega\right)$$

Since $\pi(s,t) \leq 1$ and we need to provide guarantee for arbitrarily given s and t, we can further derive that:

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

Let $\exp\left(-\frac{\lambda^2}{2+\lambda}\omega\right) = p_f$. Then we have $\omega = (2+\lambda)\cdot\frac{\log(1/p_f)}{\lambda^2}$, which finishes the proof. □

Proof of Equation 1. We first prove for the case $s \neq t$. For each in-neighbour v of node t, let X be an event that a random walk from s reaches v but does not stop at v. It is easy to verify that the probability is $\frac{(1-\alpha)\cdot\pi(s,v)}{\alpha}$. Given event X, consider an event Y that the random walk stops at node t, since it has $\frac{1}{|Out(v)|}$ probability chances to jump to t, and the probability that it terminates at t is α, then the probability of event Y under X is:

$$Pr[Y|X] = \pi(s,v)\cdot\frac{(1-\alpha)}{\alpha}\cdot\frac{\alpha}{|Out(v)|} = \pi(s,v)\cdot\frac{(1-\alpha)}{|Out(v)|}.$$

Adding them together for each in-neighbour of node t derives Equation 1. When $s = t$, α portion of the random walk will terminates at s, and hence an additional α should be added in the equation. □

Proof of Proposition 3.2. Firstly, obtain a sequence $X_1, X_2, \cdots, X_\omega$ of samples of X, and define $\hat{X} = \frac{1}{\omega}\sum_{i=1}^{\omega} X_i$. Return "yes" if $\hat{X} > \phi' \cdot r$ and "no" otherwise. According the definition of pairwise AHH query, we want to guarantee that if $\mathbb{E}[X] > (1+c)\cdot\phi'\cdot r$ we return yes, i.e., $\hat{X} > \phi'\cdot r$, and if $\mathbb{E}[X] < (1-c)\cdot\phi'\cdot r$ we return no, i.e., $\hat{X} < \phi' \cdot r$, with high probability. We next prove that with the above answering method, we can distinguish $\mathbb{E}[X] > (1+c)\cdot\phi'\cdot r$ and $\mathbb{E}[X] < (1-c)\cdot\phi'\cdot r$ with high probability. Define $Y = X/r$, $\hat{Y} = \hat{X}/r$, then $Y \in [0,1]$. First consider the case $\mathbb{E}[X] > (1+c)\cdot\phi'\cdot r$, we want to guarantee $\hat{X} > \phi'\cdot r$ with $1 - p_f$ probability. It is the same to consider $\mathbb{E}[Y] > (1+c)\cdot\phi'$,

Algorithm 3: BLOG-R Algorithm

Input: Graph G, source node s, a target set T, probability α
Output: Whether t is an AHH of s or not for each $t \in T$

1 Let $y(v) = 0$ for $v \in T$;
2 **if** $|T| > \frac{1}{\phi \cdot c_{push}}$ **then**
3 Use Monte-Carlo to calculate $\hat{\pi}(s, t)$ for each $t \in T$;
4 $y(t) \leftarrow 1$ if $\hat{\pi}(s, t) > \phi \cdot \pi(t)$ and return y;
5 Let T_i be a empty set for $i = 1$ to $\lfloor \log_2(n/\alpha) \rfloor$;
6 **for** *each* $t \in T$ **do**
7 Add t to set $T_{\lfloor \lfloor \log_2(n/\pi(t)) \rfloor \rfloor}$;
8 Let $\omega = 0$;
9 **for** $i = 1$ *to* $\lfloor \log_2(n/\alpha) \rfloor$ **do**
10 Calculate r^i_{max} according to Equation 5 where $w' = n/2^i$ and $|T_s| = |T_i|$;
11 Record the number ω_i of random walks required for bucket T_i;
12 **if** $\omega^i > \omega$ **then**
13 $\omega \leftarrow \omega_i$;
14 Sample ω random walks from s;
15 **for** $i = 1$ *to* $\lfloor \log_2(n/\alpha) \rfloor$ **do**
16 Let $r^i_{max} = \omega/\omega_i \cdot r^i_{max}$;
17 **for** $t \in T_i$ **do**
18 Invoke Algorithm 1 with t as target and r_{max} set to r^i_{max};
19 Calculate $\hat{\pi}(s, t)$ according to Equation 2;
20 **if** $\hat{\pi}(s, t) > \phi \cdot \pi(t)$ **then**
21 $y(t) \leftarrow 1$;
22 return y;

and we want to guarantee that $\hat{Y} > \phi'$ with $1 - p_f$ probability. Let $\epsilon = 1 - \phi'/\mathbb{E}[Y]$. Then, according to the Chernoff bound, we have:

$$Pr[\hat{Y} - \mathbb{E}[Y] \leq -\epsilon \cdot \mathbb{E}[Y]] \leq \exp\left(-\frac{\epsilon^2}{2}\omega \cdot \mathbb{E}[Y]\right)$$

$$\Leftrightarrow Pr[\hat{Y} \leq \phi'] \leq \exp\left(-\frac{(1 - \phi'/\mathbb{E}[Y])^2}{2}\omega \cdot \mathbb{E}[Y]\right)$$

$$\Leftrightarrow Pr[\hat{Y} \leq \phi'] \leq \exp\left(-\frac{(\mathbb{E}[Y] - \phi')^2}{2\mathbb{E}[Y]}\omega\right)$$

Let $\exp\left(-\frac{(\mathbb{E}[Y]-\phi')^2}{2\mathbb{E}[Y]}\omega\right) = p_f$, i.e., $\omega = \frac{2\mathbb{E}[Y]\cdot\log(1/p_f)}{(\mathbb{E}[Y]-\phi')^2}$. Then, we guarantee that $\hat{Y} > \phi'$ with $1 - p_f$ probability. It suffices to derive a upper bound of ω and it can be verified that $f(z) = \frac{2z\cdot\log(1/p_f)}{(z-\phi')^2}$ is monotonically decreasing when $z > \phi'$. Since $\mathbb{E}[Y] > (1 + c) \cdot \phi'$, it suffices to sample $2(\frac{1}{c^2} + \frac{1}{c})\frac{\log(1/p_f)}{\phi'}$ random walks to provide the correct answer with $1 - p_f$ probability.

Next, we consider the other case, i.e., when $\mathbb{E}[X] < (1 - c) \cdot \phi' r$, and we want to guarantee that $\hat{X} < \phi' \cdot r$ with $1 - p_f$ probability. It is the same to consider $\mathbb{E}[Y] < (1 - c) \cdot \phi'$, and guarantee that $\hat{Y} < \phi'$ with $1 - p_f$ probability. Let $\epsilon = \phi'/\mathbb{E}[Y] - 1$. According to the Chernoff bound, we have that:

$$\Pr[\hat{Y} - \mathbb{E}[Y] \geq \epsilon \cdot \mathbb{E}[Y]] \leq \exp\left(-\frac{\epsilon^2}{2 + \epsilon}\omega \cdot \mathbb{E}[Y]\right),$$

Since $\epsilon = \phi'/\mathbb{E}[Y] - 1$, it can be derived that $\mathbb{E}[Y] + \epsilon \cdot \mathbb{E}[Y] = \phi'$. Therefore, to consider the probability $\Pr[\hat{Y} - \mathbb{E}[Y] \geq \epsilon \cdot \mathbb{E}[Y]]$, it is equivalent to consider the probability $\Pr[\hat{Y} > \phi']$, and we have:

$$\Pr[\hat{Y} \geq \phi'] \leq \exp\left(-\frac{(\phi'/\mathbb{E}[Y] - 1)^2}{2 + \phi'/\mathbb{E}[Y] - 1}\omega \cdot \mathbb{E}[Y]\right)$$

$$\Leftrightarrow \Pr[\hat{Y} \geq \phi'] \leq \exp\left(-\frac{(\phi' - \mathbb{E}[Y])^2}{\phi' + \mathbb{E}[Y]}\omega\right)$$

Let $\exp\left(-\frac{(\phi'-\mathbb{E}[Y])^2}{\phi'+\mathbb{E}[Y]}\omega\right) = 1/p_f$, i.e., $\omega = \frac{(\phi'+\mathbb{E}[Y])\cdot\log(1/p_f)}{(\phi'-\mathbb{E}[Y])^2}$. Then we guarantee that $\hat{Y} < \phi'$ with $1 - p_f$ probability. Similarly, we only need to derive an upper bound of ω. Let $f'(z) = \frac{(\phi'+z)\cdot\log(1/p_f)}{(\phi'-z)^2}$. When $0 < z < \phi'$, it can be verified that $f'(z)$ is monotonically increasing. Then, by setting $z = (1 - c) \cdot \phi$, we have the maximum number of samples required to provide the guarantee. The number of random walks required is $(\frac{2}{c^2} - \frac{1}{c}) \cdot \frac{\log(1/p_f)}{\phi'}$.

Since c is a constant, it suffices to sample $O(\frac{\log(1/p_f)}{\phi'})$ random walks to guarantee both cases, which finishes the proof. □

Proof of Lemma 3.3. Firstly, if $y > w$, we know that $y + \mathbb{E}[X] > w$, and hence, we always return "yes". Next consider $y \leq \phi \cdot w$. If $w \cdot \phi - y > r$ or $c \cdot \phi \cdot w > r$, $(1 + c) \cdot \phi \cdot w - y$ exceeds r (making $y + E[X] > (1 + c) \cdot \phi \cdot w$ impossible), and we always return "no". Therefore, we only need to focus on the case when $\phi \cdot w - y \leq r$, $c \cdot \phi \cdot w \leq r$, and $\phi \cdot w \geq y$. To distinguish:

$$E[X] > (1 + c) \cdot \phi \cdot w - y$$
$$E[X] < (1 - c) \cdot \phi \cdot w - y$$

It is equivalent to distinguish:

$$E[X] > (\frac{\phi \cdot w - y}{r} + \frac{c \cdot \phi \cdot w}{r}) \cdot r$$
$$E[X] < (\frac{\phi \cdot w - y}{r} - \frac{c \cdot \phi \cdot w}{r}) \cdot r$$

By setting $\phi' = \frac{\phi \cdot w - y}{r}$ and $c' = \frac{c \cdot \phi \cdot w}{r \cdot \phi'}$, we can still use Proposition 3.2. The total number of random walks required is at most:

$$2(\frac{1}{c'^2\phi'} + \frac{1}{c'\phi'}) \cdot \log(1/p_f)$$

By replacing the values of ϕ' and c', we have that the total number of random walks is:

$$2(\frac{\phi \cdot w - y}{c^2\phi^2 \cdot w} + \frac{1}{c \cdot \phi}) \cdot \frac{r}{w} \cdot \log(1/p_f)$$

Since c is a constant, we can further have the final complexity as $O\left((\frac{\phi w - y}{\phi^2 w} + \frac{1}{\phi})\frac{r}{w}\log(1/p_f)\right)$. □

B ALGORITHM DETAILS OF BLOG-R AND BLOG-M

BLOG-R. Algorithm 3 shows the pseudo-code of our BLOG-R algorithm. We first check whether $|T| > \frac{1}{\phi \cdot c_{push}}$, and if the condition is true, BLOG-R simply runs the Monte-Carlo method and return the answer for the target set (Lines 2-4). Otherwise, BLOG-R applies the logarithmic bucketing approach to handle the query. In particular, given the target set T, it first divides the nodes in T into different buckets (Lines 5-7). Then, for each bucket, it calculates the backward threshold r^i_{max} and the number ω_i of random walks required to achieve the approximation guarantee for each node in

Algorithm 4: BLOG-M Algorithm

Input: Graph G, source set S, a target set T, probability α
Output: Whether t is an AHH of s or not for each $s \in S$, $t \in T$

1 Let $y(s, t) = 0$ for $s \in S$, $t \in T$;
2 **if** $|T| > \frac{|S|}{\phi \cdot c_{push}}$ **then**
3 **for** *each* s *in* S **do**
4 Use Monte-Carlo to calculate $\hat{\pi}(s, t)$ for each $t \in T$;
5 $y(s, t) \leftarrow 1$ if $\hat{\pi}(s, t) > \phi \cdot \pi(t)$;
6 return y;
7 Identical to Algorithm 3 Lines 5-13 except that r^i_{max} is set according to Equation 6;
8 **for** *each* $s \in S$ **do**
9 Sample ω random walks from s;
10 Identical to Algorithm 3 Lines 14-21 except that we sample random walks from each source (Line 14), and obtain $y(s, t)$ for each $s \in S$ and $t \in T$;
11 return y;

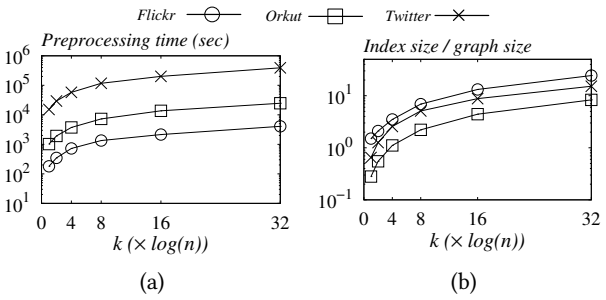


Figure 6: Preprocessing cost of LBP.

the bucket (Line 10-11). Meanwhile, it records the maximum number ω of random walks required to achieve approximation guarantee for each bucket and sample ω random walks from s (Line 12-14). Then, BLOG-R runs the backward propagation from each bucket. For bucket i, BLOG-R first tunes down r^i_{max} without compromising the approximation guarantee (Line 16), since it selects the maximum number of random walks required to start from s. With this strategy, it further reduces the backward cost from the target nodes in bucket i since r^i_{max} is set to a larger value and hence help reduces the backward cost. If the estimated PPR $\hat{\pi}(s, v)$ is above $\phi \cdot \pi(v)$, BLOG-R sets its answer to yes, and otherwise zero (Lines 20-21), and returns the answer for the entire target set in Line 22.

BLOG-M. Algorithm 4 shows the pseudo-code of the proposed BLOG-M. The key idea is similar to BLOG-R and the main difference is how to set r_{max} for each bucket. In particular, for each bucket, r_{max} is set according to Equation 6, which balances the cost of forward random walks and the backward propagation cost within the bucket. Then, BLOG-M samples random walks from each source in S. The backward propagation is similar to that of BLOG-R, i.e., applies the same r^i_{max} for the nodes bucket T_i. Finally, it obtains the estimation $\hat{\pi}(s, t)$ for each $s \in S$ and $t \in T$, checks if s is AHH of t, and returns the final answer for the query.

C EXTENDING BIPPR TO HEAVY HITTER QUERIES

In this section, we demonstrate how to extend BiPPR to answer the AHH queries. In particular, given a source s, a target t, an absolute error threshold λ, we extend BiPPR so that it provides an estimated PPR score $\hat{\pi}(s, t)$ satisfying that $|\hat{\pi}(s, t) - \pi(s, t)| < \lambda$. In BiPPR, it first proceeds the backward propagation from t with a pre-defined r_{max}, and then start random walks from s to derive the estimation of $\pi(s, t)$. According to Equation 2, we can define a random variable X as follows: it starts a random walk from s, and it stops at t, then X is $r(v, t)$, otherwise is zero. Then, the expectation of X is $\sum_{v \in V} r(v, t) \cdot \pi(s, v)$. Also, note that $r(v, t) < r_{max}$. Therefore, define $Y = X/r_{max}$, we know that Y can be bounded by $[0, 1]$, and to provide λ guarantee for PPR score, it suffices to provide λ/r_{max} guarantee for Y. Apply the similar technique as shown in Proof of Lemma 2.9, it can be proved that it suffices to sample $\frac{3r_{max} \cdot \log(1/p_f)}{\lambda^2}$ random walks. Recall that the cost of backward propagation is $O\left(\frac{\pi(t) \cdot c_{push}}{r_{max}}\right)$. Therefore, the total cost of BiPPR is:

$$O\left(\frac{\pi(t) \cdot c_{push}}{r_{max}} + \frac{3r_{max} \cdot \log(1/p_f)}{\lambda^2}\right).$$

By setting $r_{max} = \lambda \cdot \sqrt{\frac{\pi(t) \cdot c_{push}}{3 \log(1/p_f)}}$, BiPPR derives the minimized time complexity, which is $O(\frac{1}{\lambda}\sqrt{\pi(t) \cdot c_{push} \cdot \log(1/p_f)})$. By setting $\lambda = c \cdot \phi \cdot \pi(t)$, BiPPR can be extended to answer the pairwise AHH queries.

D ADDITIONAL EXPERIMENTS

In this set of experiment, we present the preprocessing cost of the LBP algorithm (ref. Section 4). Figures 6(a)-(b) demonstrate the preprocessing time and space, respectively, of LBP on three representative datasets. As we can observe from Figure 6(a), with the increase of k, i.e., the number of top-k highest in-degree nodes with pre-stored PPRs, the preprocessing time also increases. The preprocessing time is high but is still acceptable since we only need to pre-calculate once and reuse the results for handing queries. Besides, the preprocessing can be further improved with multi-core parallelization. For the preprocessing cost, we report the ratio of the index size to the graph size. As we can observe, when k reaches $32\lceil \log(n) \rceil$, the index size is around 10 times that of the graph size. Recall that the worst case time complexity decreases by more than an order of magnitude when k reaches $32\lceil \log(n) \rceil$, this demonstrates the effectiveness of the LBP algorithm.