Fast Inbound Top-K Query for Random Walk with Restart

Chao Zhang Shan Jiang Yucheng Chen Yidan Sun Jiawei Han

Dept. of Computer Science, University of Illinois at Urbana-Champaign, IL, USA
{czhang82, sjiang18, ychen233, ysun69, hanj}@illinois.edu

Abstract. Random walk with restart (RWR) is widely recognized as one of the most important node proximity measures for graphs, as it captures the holistic graph structure and is robust to noise in the graph. In this paper, we study a novel query based on the RWR measure, called the *inbound top-k* (`Ink`) query. Given a query node q and a number k, the `Ink` query aims at retrieving k nodes in the graph that have the largest weighted RWR scores *to* q. `Ink` queries can be highly useful for various applications such as traffic scheduling, disease treatment, and targeted advertising. Nevertheless, none of the existing RWR computation techniques can process the `Ink` query efficiently in large graphs. We propose two algorithms, namely SQUEEZE and RIPPLE, both of which can accurately answer the `Ink` query in a fast and incremental manner. To identify the top-k nodes, SQUEEZE iteratively performs matrix-vector multiplication and estimates the lower and upper bounds for all the nodes in the graph. RIPPLE employs a more aggressive strategy by only estimating the RWR scores for the nodes falling in the *vicinity* of q, the nodes outside the vicinity do not need to be evaluated because their RWR scores are propagated from the boundary of the vicinity and thus upper bounded. RIPPLE incrementally expands the vicinity until the top-k result set can be obtained. Our extensive experiments on real-life graph data sets show that `Ink` queries can retrieve interesting results, and the proposed algorithms are orders of magnitude faster than state-of-the-art method.

1 Introduction

Graphs have long been considered as one of the most important structures that can naturally model numerous real-life data objects (*e.g.*, the Web, social network, protein-protein interaction network). In most graph-related applications, it is fundamental to quantify node-to-node structural proximity. Among existing structural proximity measures, *random walk with restart* (RWR) is recognized as one of the most important, and has been widely adopted in Web search [14], item recommendation [11], link prediction [12], graph clustering [1], and many other tasks. Compared with other proximity measures like shortest path, RWR enjoys the nice property of capturing the holistic graph structure and being robust to noise in the graph.

To date, much research effort has been devoted to RWR, including its efficient computation ([5], [15], [4], [6], [18], [7], [13]), top-k search ([6], [9], [2], [16]), and various mining tasks underpinned by RWR ([12], [1], [11]). However, none of them have considered a fundamental task that arises in many graph-related applications, which is to

determine the source nodes that *have a large amount of information flowing to* a given query node. To illustrate, consider a traffic flow network shown in Figure 1. Assume severe traffic congestion occurs at node q every day, then the following question is key to improving traffic scheduling and road network design: how do we find the nodes from which the traffic tends to flow into q and cause the congestion problem? Using the RWR measure, the node c is likely to be identified as a major source that causes congestion at q. Even though c is not the direct in-neighbor of q, there are many short paths from c to q. Given that c is a busy transportation hub, a large number of vehicles leaving from c tend to gather at q.

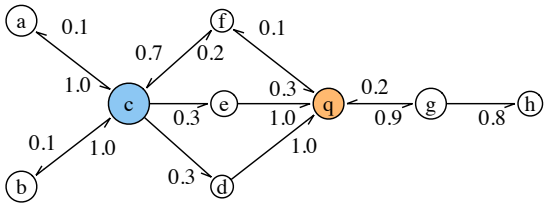


Fig. 1. An example traffic flow network. Each node is a road intersection, and the node size denotes the daily traffic volume at the intersection. Each edge is a road segment, and the attached number denotes the proportion of the traffic moving along that edge from a specific node.

We propose a novel query named the *inbound top-k* (`Ink`) query, which seeks to identify the nodes that have *a large amount of information flowing to* a query node based on RWR. Consider a query node q in a graph G. For any other node u in G, let $r_{u \rightsquigarrow q}$ be the RWR from u to q. Additionally, each node u has a nonnegative weight w_u.[1] Given G, q, and an integer k, the `Ink` query aims to find k nodes in G that have the largest scores in terms of $w_u \cdot r_{u \rightsquigarrow q}$.

The `Ink` query can be highly useful for a wide spectrum of applications besides traffic flow analysis. Think of a protein-protein interaction (PPI) network wherein each node is a protein, and a directed edge indicates one protein has a signal transduction to another protein to cause its formation or mutation. The signal transduction between proteins is essential to many biological processes and diseases (*e.g.*, Parkinson's disease, cancer). Querying by the characteristic protein of a disease, the `Ink` query can identify the top-k proteins that are most likely to cause the formation of the query protein. Another example application is *targeted advertising*. In an online social network like Facebook, suppose a company (*e.g.*, Walmart) wants to place advertisements on its Facebook page. With the `Ink` query, that company can easily identify the top-k Facebook users that are most likely to visit its page. By statistically analyzing the profiles and preferences of these users, the company can adapt the advertising content to attract potential customers more effectively.

To the best of our knowledge, no existing methods can *accurately* and *efficiently* answer `Ink` queries. First, methods ([5], [15], [13], [18]) have been proposed to compute the approximate RWR between any two nodes, with an error bound ϵ. However, it is hard to pre-specify a proper ϵ for an ad-hoc query node q, because a pre-specified ϵ may be either too coarse to generate the correct top-k results, or too fine to avoid unnecessary computation. Second, the k-dash method [6] can compute the exact RWR between

[1] For instance, in our traffic flow example, w_u is the average daily traffic volume at node u.

any two nodes. However, it uses matrix LU decomposition as a pre-computation step, which has a time complexity of $O(n^3)$ and thus is prohibitively expensive for large graphs. Even assuming the LU decomposition is done, later we will see, it is costly and unnecessary to compute the RWR scores from all nodes to the query node q in order to answer `Ink` queries. Third, techniques ([15], [9], [4], [8], [7]) has recently been reported to process what we call *outbound top-k* queries, *i.e.*, which k nodes have the largest RWR if we start the random walk *from* node q? These techniques mostly use the branch-and-bound strategy to prune the search space, but the lower and upper bounds derived for the outbound top-k query cannot be easily adapted for the `Ink` query.

To efficiently answer the `Ink` query, we propose two branch-and-bound methods. Our first method, called SQUEEZE (Section 3) does not directly compute the exact RWR to q for each node in the graph, but maintains a lower bound and an upper bound. It then incrementally refines the bounds by performing matrix-vector multiplication. We prove that the error decreases exponentially as the iterative process continues, and thus the top-k results can be determined after a few number of iterations. Our second method, called RIPPLE (Section 4), is an even more efficient algorithm and thus suitable for extremely large graphs. Compared with SQUEEZE, RIPPLE leverages locality to gain significant performance improvement. The key observation is that the nodes falling in the vicinity of q tend to have large RWR scores to q. Hence, RIPPLE maintains a dynamic vicinity of q and estimates the RWR scores only for the nodes inside the vicinity. The outside nodes do not need to be evaluated because their scores are propagated from the boundary of the vicinity and thus upper bounded. RIPPLE progressively expands the vicinity, and refines the error bounds until the result set can be correctly identified.

Our theoretical analysis showes that both SQUEEZE and RIPPLE, without any pre-computation, can accurately answer the `Ink` queries in a fast and incremental manner. In addition, we have conducted extensive experiments on real-life graph data sets (Section 5). The results demonstrate the `Ink` query can retrieve interesting results. Meanwhile, SQUEEZE and RIPPLE outperform state-of-the-art method by orders of magnitude in efficiency.

2 Preliminaries

In this section, we present some preliminaries for the `Ink` query. Table 1 lists the notations used throughout this paper.

Table 1. Notations used in the paper.

G	A graph $G = (V, E)$.	O_u	The set of u's out-neighbors.
n	The number of nodes in G.	$\mathbf{P}$	The row normalized transition matrix for G.
m	The number of edges in G.	c	The restart probability ($0 < c < 1$).
w_i	The weight of a node $i \in V$.	$\mathbf{e}_u$	$n \times 1$ vector, 1 for u's element and 0 for the others.
w_{ij}	The weight of an edge (i, j).	$\mathbf{r}_u$	The RWR score vector for the walk *from* u.

2.1 Problem Description

Definition 1 (Transition Matrix). *For a node $i \in V$, let $d_i = \sum_{j=1}^{n} w_{ij}$ be the total out-degree of i. The transition matrix of G is an $n \times n$ matrix $\mathbf{P} = [p_{ij}]_{n \times n}$ where $p_{ij} = w_{ij}/d_i$ if $(i, j) \in E$ and 0 otherwise.*

Based on the definition of transition matrix, the *random walk with restart (RWR)* process is described as follows. Consider a surfer who starts RWR from the node $x_0 = u$. Suppose the surfer is at node $x_t = i$ at step t, she returns to u with probability c and continues surfing with probability $1 - c$. If she continues the surfing, she randomly moves to i's neighbor j with probability p_{ij}. The stationary distribution $\mathbf{r}_u$ of such a process, *i.e.*, the RWR scores of all the nodes in V, is the solution to the equation:

$$\mathbf{r}_u = (1 - c)\mathbf{P}^{\mathrm{T}}\mathbf{r}_u + c\mathbf{e}_u. \tag{1}$$

In $\mathbf{r}_u$, the element $\mathbf{r}_u(v)$ denotes the RWR score from u to v, namely $r_{u \rightsquigarrow v}$. Given a query node $q \in V$, a restart probability c, and an integer k, **the `Ink` query** aims to find a set $S \subseteq V$ such that: (1) $|S| = k$; and (2) $\forall u \in S, \forall v \in V - S$, $w_u \mathbf{r}_u(q) \geq w_v \mathbf{r}_v(q)$.

2.2 Naïve Methods

In this subsection, we describe two naïve methods for anwering `Ink` queries, and discuss why they are not satisfactory enough.

Power: Although directly solving Equation 1 costs $O(n^3)$, the power iteration can produce an approximate solution with time complexity $O(tm)$ where t is the number of iterations. Accordingly, a naïve solution, named Power, can answer the `Ink` query in two steps: (1) it computes $\mathbf{r}_u$ for every node $u \in V$ using the power iteration; and (2) it selects k nodes with the largest weighted RWRs to q.

LU: Another solution, named LU, is adapted from the *k-dash* method proposed by Fujiwara *et al.* [6]. As the solution of Equation 1 is $\mathbf{r}_u = c(\mathbf{I} - (1 - c)\mathbf{P}^{\mathrm{T}})^{-1}\mathbf{e}_u$, they perform LU decomposition on the matrix $\mathbf{W} = \mathbf{I} - (1 - c)\mathbf{P}^{\mathrm{T}} = \mathbf{L}\mathbf{U}$ in an offline stage, and store $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$ beforehand. As $\mathbf{r}_u = c\mathbf{W}^{-1}\mathbf{e}_u = c\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{e}_u$, the exact RWR score between any two nodes can be computed in $O(n)$ time based on the matrices $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$. Accordingly, LU does not need to compute the entire proximity matrix to answer the `Ink` query. Instead, it computes only one row that corresponds to the RWR scores from all the nodes to the query node q. Once the n RWR scores are obtained, the top-k nodes can be easily obtained.

Remark. For every $u \in V$, the Power method needs to compute u's RWR scores to all the nodes in V, leading to a time complexity $O(tmn)$, which is intolerable for large graphs. The LU method can directly compute the exact RWR scores from all nodes to the query node based on offline matrix decomposition. However, the time complexity of the on-line retrieval phase is $O(n^2 + n \log k)$, still time-consuming for large graphs. Later we will see, it is actually unnecessary and wasteful to compute the RWRs from all the nodes to q. Moreover, note that matrix LU decomposition has a time complexity of $O(n^3)$ and thus is prohibitively expensive for large graphs. Finally, the matrix $\mathbf{W}$ is dependent on the restart probability c. If the user launches an `Ink` query with a different c, the precomputed matrices $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$ become useless and need to be recomputed.

2.3 Overview of Squeeze and Ripple

Before presenting the SQUEEZE and RIPPLE methods, we first analyze the relations of the RWR scores from all the nodes to the query node q. We start with the *Decomposition Theorem* proposed by Jeh and Windom [10].

Theorem 1. *Given a node u, and O_u, the set of u's out-neighbors, the RWR proximity vector from u satisfies* $\mathbf{r}_u = (1-c)\sum_{v \in O_u} p_{uv}\mathbf{r}_v + c\mathbf{e}_u$.

Theorem 1 says that, the RWR vector of u can be derived by linearly combining the RWR vectors of u's out-neighbors, with extra emphasis on u itself. For any node $u \in V$, we have the RWR score from u to q computed as:

$$\mathbf{r}_u(q) = \begin{cases} (1-c)\sum_{v \in O_u} p_{uv}\mathbf{r}_v(q) & \text{if } u \neq q \\ (1-c)\sum_{v \in O_u} p_{uv}\mathbf{r}_v(q) + c & \text{if } u = q. \end{cases} \tag{2}$$

By writing down the decomposition for every node u in V according to Equation 2, we obtain a linear system $\mathbf{x}_q$ that consists of n variables. Specifically, letting $\mathbf{x}_q$ be an $n \times 1$ vector such that $\mathbf{x}_q(u) = \mathbf{r}_u(q)$ is the RWR score from u to q, and letting $\mathbf{A} = (1-c)\mathbf{P}$, then

$$\mathbf{x}_q = \mathbf{A}\mathbf{x}_q + c\mathbf{e}_q. \tag{3}$$

An intuitive idea to answer `Ink` query is to perform power iteration over Equation 3 and obtain a good "enough" approximation of $\mathbf{x}_q$. Based on this intuition, our first method SQUEEZE iteratively performs matrix-vector multiplication and provides an analytical error bound after each iteration. We prove that the error shrinks at an exponential rate as the iteration proceeds, hence SQUEEZE can prune the unqualified nodes quickly and retrieve the top-k results after a small number of iterations.

Though SQUEEZE is simple, performing matrix-vector multiplication over the whole graph can be costly if the graph is extremely large. Our second method RIPPLE addresses this problem by leveraging the locality of RWR. The key observation is that the nodes around q tend to have large RWR scores. As such, RIPPLE employs a local update strategy, which maintains a vicinity around q and evaluates RWR only for the nodes inside. According to Equation 3, the RWR scores of the nodes outside the vicinity are propagated from the boundary of the vicinity and thus upper bounded. By progressively pushing the boundary of the vicinity, the estimations for the inside nodes become more accurate, and the upper bound for the outside nodes becomes tighter. Finally, RIPPLE terminates the vicinity expansion once the top-k results can be correctly identified.

3 The Squeeze Algorithm

In this section, we describe the details of SQUEEZE. As aforementioned, the key idea of SQUEEZE is to iterate over Equation 3, and analyze the estimation errors on-the-fly. To begin with, we define the lower bound relation between two vectors.

Definition 2 (Lower Bound Vector). *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be two* $n \times 1$ *vectors.* $\mathbf{x}$ *is a lower bound vector of* $\mathbf{y}$ *if* $\forall 1 \leq i \leq n$, $\mathbf{x}(i) \leq \mathbf{y}(i)$, *denoted as* $\mathbf{x} \prec \mathbf{y}$.

SQUEEZE starts with the zero vector $\mathbf{x}_q^{(0)} = \mathbf{0}$, which serves as a lower bound vector for $\mathbf{x}_q$, the solution to Equation 3. Then it iteratively updates the lower bound vector according to the following equation:

$$\mathbf{x}_q^{(i+1)} = \mathbf{A}\mathbf{x}_q^{(i)} + c\mathbf{e}_q. \tag{4}$$

In the following, we prove: (1) each iteration produces a tighter lower bound vector, *i.e.*, $\mathbf{x}_q^{(i+1)}$ will be closer to $\mathbf{x}_q$, and (2) $\mathbf{x}_q^{(i)}$ finally converges to $\mathbf{x}_q$.

Theorem 2. *Let* $\mathbf{x}_q^{(0)} = \mathbf{0}$ *and* $\mathbf{x}_q^{(i+1)} = \mathbf{A}\mathbf{x}_q^{(i)} + c\mathbf{e}_q$. *It is ensured* $\forall i \geq 0, \mathbf{x}_q^{(i)} \prec \mathbf{x}_q^{(i+1)} \prec \mathbf{x}_q$; *and* $\mathbf{x}_q^{(i)} = \mathbf{x}_q$ *when* $i \to \infty$.

Proof. (1) Given $\mathbf{x}_q^{(0)} = \mathbf{0}$ and Equation 4, we have $\mathbf{x}_q^{(1)}(u) = c \cdot \mathbb{I}_{\{u=q\}}(u)$, where $\mathbb{I}$ is the indicator function. Clearly, $\mathbf{x}_q^{(0)} \prec \mathbf{x}_q^{(1)}$. Further, if $\mathbf{x}_q^{(i-1)} \prec \mathbf{x}_q^{(i)}$, then $\forall u$,

$$\mathbf{x}_q^{(i+1)}(u) - \mathbf{x}_q^{(i)}(u) = (1-c) \sum_{v \in O_u} p_{uv} \left[\mathbf{x}_q^{(i)}(u) - \mathbf{x}_q^{(i-1)}(u) \right] \geq 0.$$

(2) It is clear that $\mathbf{x}_q^{(0)} \prec \mathbf{x}_q$. Suppose $\mathbf{x}_q^{(i)} \prec \mathbf{x}_q$, then

$$\mathbf{x}_q^{(i+1)}(u) \leq (1-c) \sum_{v \in O_u} p_{uv} \mathbf{x}_q(v) + c \cdot \mathbb{I}_{\{u=q\}}(u) = \mathbf{x}_q(u).$$

To prove $\lim_{i \to \infty} \mathbf{x}_q^{(i)} = \mathbf{x}_q^{(i)}$, note that the spectral radius of $\mathbf{A}$ satisfies $\rho(\mathbf{A}) \leq 1 - c < 1$. Then $\lim_{i \to \infty} \mathbf{x}_q^{(i)} = \sum_{i=0}^{\infty} \mathbf{A}^i \mathbf{e}_q = c(\mathbf{I} - \mathbf{A})^{-1} \mathbf{e}_q$, which is the solution of Equation 4. ■

Theorem 2 tells us that the power iteration over Equation 4 produces a tighter lower bound after each iteration, and finally converges to the exact value of $\mathbf{x}_q$. Below, we proceed to analyze the RWR upper bound after each iteration.

Theorem 3. $\forall u \in V, \mathbf{x}_q(u) \leq \mathbf{x}_q^{(i)}(u) + (1-c)^i$.

Proof. $\forall i > 0, \mathbf{x}_q^{(i+1)} - \mathbf{x}_q^{(i)} = \mathbf{A}(\mathbf{x}_q^{(i)} - \mathbf{x}_q^{(i-1)})$. Accordingly,

$$\|\mathbf{x}_q^{(i+1)} - \mathbf{x}_q^{(i)}\| \leq \|\mathbf{A}\| \cdot \|\mathbf{x}_q^{(i)} - \mathbf{x}_q^{(i-1)}\| = (1-c) \cdot \|\mathbf{x}_q^{(i)} - \mathbf{x}_q^{(i-1)}\|.$$

Recursively applying the above inequality gives us $\|\mathbf{x}_q^{(i+1)} - \mathbf{x}_q^{(i)}\| \leq (1-c)^i \|\mathbf{x}_q^{(1)} - \mathbf{x}_q^{(0)}\| = (1-c)^i c$. Moreover, $\forall m > i$,

$$\|\mathbf{x}_q^{(m)} - \mathbf{x}_q^{(i)}\| = \|\sum_{j=i}^{m-1} (\mathbf{x}_q^{(j+1)} - \mathbf{x}_q^{(j)})\| \leq \sum_{j=i}^{m-1} \|\mathbf{x}_q^{(j+1)} - \mathbf{x}_q^{(j)}\|$$

$$\leq (1-c)^i c \sum_{j=0}^{m-i-1} (1-c)^j = (1-c)^i c \frac{1-(1-c)^{m-i}}{1-(1-c)}.$$

By setting $m \to \infty$, we have $\|\mathbf{x}_q - \mathbf{x}_q^{(i)}\| \leq (1-c)^i$. ■

Theorem 2 and 3 guarantee that after iteration i, $\forall u \in V$, $\mathbf{x}_q^{(i)}(u) \leq \mathbf{x}_q(u) \leq \mathbf{x}_q^{(i)}(u) + (1-c)^i$. Better still, the gap $(1-c)^i$ between the lower and upper bounds decreases exponentially as the iteration proceeds, which allows us to quickly identify the top-k results. Algorithm 1 sketches SQUEEZE. As shown, we first set $\mathbf{x}_q = \mathbf{0}$, and initialize a candidate set R that consists of all the nodes in V. Then we gradually refine $\mathbf{x}_q$ using power iteration (line 4), and select the k-th largest weighted RWR as the threshold τ (line 5). All the nodes whose weighted RWRs are smaller than τ can be safely pruned (lines 7-9). SQUEEZE terminates when the candidate set R contains only k nodes.

Algorithm 1: The SQUEEZE algorithm.

Input: query node q, number k, graph $G = (V, E)$, restart probability c
Output: the top-k result set R

```
1 R ← V, i ← 0, x_q ← 0;
2 Construct the transition matrix P;
3 while |R| > k do
4     x_q ← A x_q + c e_q;
5     τ ← k-th largest score in terms of w_u · x_q(u);
6     i ← i + 1;
7     foreach u ∈ R do
8         if w_u · [x_q + (1 − c)^i] < τ then
9             Remove u from R;
```

4 The Ripple Algorithm

In this section, we present the RIPPLE method, which employs a local update strategy to efficiently process `Ink` queries.

4.1 Algorithm Sketch

Given a query node q, we use N_q to denote a set of nodes falling in the vicinity of q, and F_q to denote the nodes falling outside, *i.e.*, $F_q = V - N_q$. Further, we call node $u \in N_q$ a *boundary node* if there exists a node $v \in F_q$ such that v is an in-neighbor of u, and use B_q to denote the set of boundary nodes. The key insight of RIPPLE is that the nodes close to q tend to have large RWR scores. Hence, starting from a small vicinity around q, RIPPLE estimates the RWRs for only the nodes in the vicinity. For the outside nodes, RIPPLE maintains one generic upper bound for them. As the vicinity is gradually expanded, the RWR estimations for the inside nodes as well as the upper bound for the outside node become more and more accurate. The expansion terminates when the estimations are accurate enough to produce the top-k results.

Algorithm 2 gives a sketch of RIPPLE. As shown, we initialize N_q and B_q to $\{q\}$, and iteratively select at most s boundary nodes with the largest RWRs (lines 3-6), where s is a pre-specified parameter of RIPPLE. Later we will see, the rationale of selecting the high-score boundary nodes is that such nodes determine the estimation error for the

nodes in N_q, as well as the RWR upper bound for the nodes in F_q. We expand N_q by incorporating the in-neighbors of the selected nodes (lines 7-8). After each expansion, we iterate over the nodes in N_q for t times to refine their RWR estimations (lines 10-12). The RWR scores for all the nodes in F_q are set to 0 and do not need to be computed. Once the update operation is done, we select the k-th largest weighted RWR as the threshold τ, and prune the nodes whose upper bound scores are smaller than τ (lines 14-17). Such a process repeats until there are only k nodes left in R.

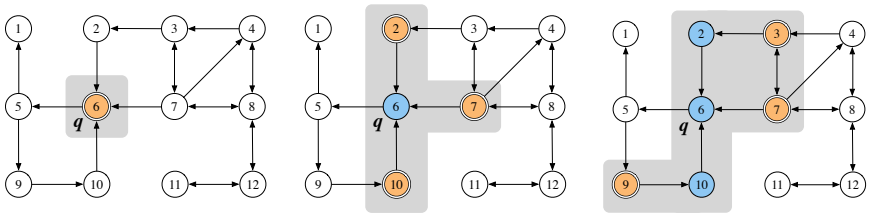
Algorithm 2: The RIPPLE algorithm.

Input: query node q, number k, graph $G = (V, E)$, restart probability c, number of to-expand nodes s, number of iterations t after expansion
Output: the top-k result set R

1 $R \leftarrow V$, $N_q \leftarrow \{q\}$, $B_q \leftarrow \{q\}$, $\underline{\mathbf{x}}_q \leftarrow \mathbf{0}$;
2 **while** $|R| > k$ **do**
3 **if** $|B_q| \geq s$ **then**
4 $E \leftarrow s$ nodes in B_q with the largest RWRs;
5 **else**
6 $E \leftarrow B_q$;
7 **foreach** $u \in E$ **do**
8 Add u's in-neighbors into N_q;
9 Update B_q;
10 **for** $i = 1$ **to** t **do**
11 **foreach** $u \in N_q$ **do**
12 $\underline{\mathbf{x}}_q(u) = (1-c) \sum_{v \in O_u} p_{uv} \underline{\mathbf{x}}_q(u) + c \cdot \mathbb{I}_{\{u=q\}}(u)$;
13 $\tau \leftarrow k$-th largest weighted RWR for the nodes in N_q;
14 **foreach** $u \in R$ **do**
15 $\overline{\mathbf{x}}_q(u) \leftarrow$ the RWR upper bound;
16 **if** $w_u \cdot \overline{\mathbf{x}}_q(u) < \tau$ **then**
17 Remove u from R;
18 **return** R;

Figure 2 shows a concrete example of RIPPLE. Suppose $c = 0.2$, $s = 2$, and $t = 2$. First, the vicinity and boundary node sets are set to $N_q = B_q = \{q\}$. In the first round, q is the only node B_q. Hence, we expand q and obtain $N_q = \{2, 6, 7, 10\}$ and $B_q = \{2, 7, 10\}$. After the expansion, starting from $\mathbf{x}_q = \mathbf{0}$, RIPPLE updates the RWR for the nodes in N_q using 2 iterations, and obtains $\mathbf{x}_q(2) = 0.16, \mathbf{x}_q(6) = 0.2, \mathbf{x}_q(7) = 0.04, \mathbf{x}_q(10) = 0.16$. In the second round, RIPPLE expands node 2 and 10, and derives $N_q = \{2, 3, 6, 7, 9, 10\}$ and $B_q = \{3, 7, 9\}$. With the previous $\mathbf{x}_q$, RIPPLE updates the new N_q using 2 iterations, and obtains $\mathbf{x}_q(2) = 0.16, \mathbf{x}_q(3) = 0.08, \mathbf{x}_q(6) = 0.2, \mathbf{x}_q(7) = 0.056, \mathbf{x}_q(9) = 0.128, \mathbf{x}_q(10) = 0.16$. The expansion process continues until the top-k nodes are obtained.

Several questions remain to be answered for Algorithm 2: (1) how do we compute the lower and upper bounds for the nodes in N_q and F_q? and (2) what is the reason of selecting high-score boundary nodes when expanding N_q? In what follows, we answer these questions in detail.



(a) Initialization. (b) One round of expansion. (c) Two rounds of expansion.

Fig. 2. Illustration of the RIPPLE algorithm. The nodes in the gray area are the vicinity nodes, and the double-ringed ones are the boundary nodes.

4.2 The Lower Bound

We first prove the RWR estimation is a lower bound when we set the RWR scores of the nodes in F_q to 0 and propagate the RWR scores only among the nodes in N_q.

Theorem 4. *Let $\underline{\mathbf{x}}_q$ be the solution to the equation $\underline{\mathbf{x}}_q = \mathbf{W}\underline{\mathbf{x}}_q + c\mathbf{e}_q$, where $\mathbf{W}$ is constructed from $\mathbf{A}$ by setting the rows of nodes in F_q to all zeros, then $\underline{\mathbf{x}}_q \prec \mathbf{x}_q$.*

Proof. The power method gives us $\underline{\mathbf{x}}_q = \lim_{i\to\infty} c\sum_{j=1}^{i-1}\mathbf{W}^j\mathbf{e}_q$ and $\mathbf{x}_q = \lim_{i\to\infty} c\sum_{j=1}^{i-1}\mathbf{A}^j\mathbf{e}_q$. It suffices to prove $\forall i \geq 1, \mathbf{W}^i\mathbf{e}_q \prec \mathbf{A}^i\mathbf{e}_q$, which can be easily proved by induction. ■

4.3 The Upper Bound

We proceed to analyze the RWR upper bounds. Let $M = \max_{u\in B_q}\mathbf{x}_q(u)$. Lemma 1 and Lemma 2 show that M determines the upper bound for the nodes in both F_q and N_q.

Lemma 1. $\forall u \in F_q, \mathbf{x}_q(u) \leq (1-c)M$.

Proof. When iterating over Equation 3 with $\mathbf{x}_q^{(0)} = \mathbf{0}$, Theorem 2 ensures $\forall u \in B_u, \forall i \geq 0, \mathbf{x}_q^{(i)}(u) \leq \mathbf{x}_q(u) \leq M$. Consider any node $u \in F_q$, when $i = 0$, $\mathbf{x}_q^{(0)}(u) = 0 \leq (1-c)M$ clearly holds. $\forall i \geq 0$, $\forall v \in F_q, \mathbf{x}_q^{(i)}(v) \leq (1-c)M$; and $\forall v \in B_q, \mathbf{x}_q^{(i)}(v) \leq M$, hence $\mathbf{x}_q^{(i+1)}(u) \leq (1-c)M$. ■

Lemma 2. $\forall u \in N_q, \mathbf{x}_q(u) \leq \underline{\mathbf{x}}_q(u) + (1-c)^2M$.

Proof. Let $\mathbf{d}_q = \mathbf{x}_q - \underline{\mathbf{x}}_q$. $\forall u \in N_q$, it suffices to prove $\mathbf{d}_q(u) \leq (1-c)^2M$. Consider an $n \times 1$ vector $\mathbf{r}_F$, where the entries of the nodes in F_q are set to their accurate RWR scores, and the entries of the nodes in N_q are set to zeros. Then $\mathbf{d}_q = \mathbf{W}\mathbf{d}_q + \mathbf{r}_F$. By setting $\mathbf{d}_q^{(0)} = \mathbf{r}_F$ and using power iteration, we have $\mathbf{d}_q = \lim_{t\to\infty}\mathbf{d}_q^{(t)}$. Note that $\forall u \in V, \mathbf{d}_q^{(0)}(u) \leq (1-c)M$. The induction ensures $\lim_{t\to\infty}\mathbf{d}_q^{(t)}(u) \leq (1-c)^2M$. ■

Lemma 2 provides a generic upper bound for the nodes in N_q. In the following, we derive a tighter upper bound for these nodes by introducing the concept of *outward hop*.

Definition 3 (Outward Hop). *For a node $u \in N_q$, the outward hop of u, denoted as $\texttt{Hop}(u)$, is the minimum number of steps that takes u to any node in F_q.*

Lemma 3. *Given a node $u \in N_q$, let $\texttt{Hop}(u) = h$, then $\mathbf{x}_q(u) - \underline{\mathbf{x}}_q(u) \leq (1-c)^{h+1}M$.*

Proof. The RWR error for the nodes in N_q is propagated layer by layer. Let $\Gamma_i \subseteq N_q$ be the set of nodes that have outward hop i. By Lemma 1, it is ensured that $\forall u \in \Gamma_1, \mathbf{d}_q(u) \leq (1-c)^2 M$. Consider any node $u \in \bigcup_{i \geq 2} \Gamma_i$, all of u's out-neighbors belong to N_q (not including any node in F_q), hence $\mathbf{d}_q(u) \leq (1-c)^3 M$. Now we proceed to consider $\forall u \in \bigcup_{i \geq 3} \Gamma_i$, u's out-neighbors belong to $\bigcup_{i \geq 2} \Gamma_i$. Hence, $\forall u \in \Gamma_3$, $\mathbf{d}_q(u) \leq (1-c)^4 M$. Continuing the above process, the proof is completed. ■

Lemma 1 and 3 give us the RWR upper bounds for the nodes in F_q and N_q in terms of M. Nevertheless, the problem is M is not known as RIPPLE only has the RWR lower bound for the nodes in N_q. Assume RIPPLE performs t power iterations over N_q to obtain an approximate vector $\underline{\mathbf{x}}_q^{(t)}$, we now analyze the relationship between M and $\underline{\mathbf{x}}_q^{(t)}$, and discuss how to compute the upper bounds based on $\underline{\mathbf{x}}_q^{(t)}$.

Lemma 4. *Let $\Delta_N = \max_{u \in V} \left[\underline{\mathbf{x}}_q^{(1)}(u) - \underline{\mathbf{x}}_q^{(0)}(u)\right]$, $\underline{M}^{(t)} = \max_{u \in V} \underline{\mathbf{x}}_q^{(t)}(u)$. We have*

$$M \leq \frac{1}{2c - c^2} \left[\underline{M}^{(t)} + (1-c)^t \cdot \Delta_N / c\right].$$

Proof. Similar to Theorem 3, it can be shown $\forall u \in V$, $\underline{\mathbf{x}}_q(u) - \underline{\mathbf{x}}_q^{(t)}(u) \leq (1-c)^t \cdot \Delta_N / c$. Let $\underline{M} = \max_{u \in B_q} \underline{\mathbf{x}}_q(u)$ and $w \in B_q$ be the node corresponding to $\underline{M}$. Then $\underline{M} - \underline{\mathbf{x}}_q^{(t)}(w) \leq (1-c)^t \cdot \Delta_N / c$. Since $\underline{\mathbf{x}}_q^{(t)}(w) \leq \underline{M}^{(t)}$, we have

$$\underline{M} - \underline{M}^{(t)} \leq \underline{M} - \underline{\mathbf{x}}_q^{(t)}(w) \leq (1-c)^t \cdot \Delta_N / c. \tag{5}$$

Further let $v \in B_q$ be the node corresponding to M. By Lemma 2, we know $M - \underline{\mathbf{x}}_q(v) \leq (1-c)^2 M$, which gives $M \leq \underline{\mathbf{x}}_q(v)/(2c - c^2)$. Also, as $\underline{\mathbf{x}}_q(v) \leq \underline{M}$, we have $M \leq \underline{M}/(2c - c^2)$. By combining this with Equation 5, the proof is completed. ■

Theorem 5. *$\forall u \in F_q$, we have*

$$\mathbf{x}_q(u) \leq \frac{1-c}{2c - c^2} \left[\underline{M}^{(t)} + (1-c)^t \cdot \Delta_N / c\right].$$

Proof. The claim is obvious from Lemma 1 and 4. ■

Theorem 6. *Given a node $u \in N_q$, let $\texttt{Hop}(u) = h$, then*

$$\mathbf{x}_q(u) \leq \underline{\mathbf{x}}_q^{(t)}(u) + \frac{(1-c)^{h+1}}{2c - c^2} \cdot \underline{M}^{(t)} + \frac{(1-c)^{h+t+1} + (2c - c^2)(1-c)^t}{2c^2 - c^3} \cdot \Delta_N$$

Proof. By Lemma 3, we have $\mathbf{x}_q(u) \leq \underline{\mathbf{x}}_q(u) + (1-c)^{h+1} M$. Moreover, similar to Theorem 3, it can be shown $\underline{\mathbf{x}}_q(u) \leq \underline{\mathbf{x}}_q^{(t)}(u) + (1-c)^t \cdot \Delta_N / c$. Combining the above with Lemma 4, the proof is completed. ■

5 Experiments

In this section, we evaluate the empirical performance of the proposed methods. All algorithms were implemented in JAVA and the experiments were conducted on a machine with Intel Xeon E5-2680 and 64GB memory.

5.1 Experimental Setup

Data Sets. Our experiments are based on two real graph data sets. Our first data set, referred to as 4SQ, is collected from Foursquare during a three-month period. The 4SQ data set consists of the check-in histories of 14,909 users living in New York. In 4SQ, each node is a place, and the node weight is set to the total number of visitors to reflect the popularity of the place. Meanwhile, there is a directed edge between two places if the check-ins at the two places occur within 3 hours. The weight of the edge is the number of users whose check-in history matches the transition. The 4SQ data set contains 48,564 nodes and 123,452 edges in total. The second data set, referred to as Wiki, is extracted from the Wikipedia graph. We have removed the non-English Wikipedia pages as well as the noisy pages that have less than 3 in-links. In the result Wiki data set, each node is a Wikipedia page, and the node weight is set to the number of in-links to reflect the importance of that page. Each edge is a directed link from one Wikipedia page to another, and all the edges have an equal weight. There are totally 4,382,715 nodes and 102,260,837 edges in the Wiki data set.

Compared Method. We described two naïve methods in Section 2.2, namely Power and LU. However, the time cost of Power is too expensive for our used data sets. Hence, we use the LU method for comparison in our experiments. LU involves an offline stage that performs matrix decomposition, and an on-line stage that retrieves the top-k results, we only include the on-line retrieval time when measuring the performance of LU.

5.2 Illustrating Cases

In this subsection, we issue several test `Ink` queries on our data sets, and compare the results retrieved by the `Ink` query and those by the outbound top-k query [6].

Table 2. A Comparison of the Inbound and Outbound Top-k Queries on 4SQ ($\mathbf{c} = 0.15$ $k = 5$).

Query	Inbound Top-5 Results		Outbound Top-5 Results	
	Rank	Place Name	Rank	Place Name
Yankee Stadium	1	Yankee Tavern	1	The Metropolitan Museum of Art
	2	Stan's Sports Bar	2	Madison Square Garden
	3	Billy's Sports Bar	3	The Central Park
	4	New York Penn Station	4	Grand Central Terminal
	5	Grand Central Terminal	5	Brooklyn Museum
Columbia University	1	Morningside Park	1	Central Park
	2	Seeley Mudd Hall	2	116th St/Columbia University MTA Subway
	3	Whole Foods Grocery	3	Newark Liberty International Airport
	4	Dinosaur Bar-B-Que	4	Grand Central Terminal
	5	Starbucks	5	Lincoln Tunnel

Table 2 shows the inbound and outbound top-5 results for the queries "Yankee Stadium" and "Columbia University" on 4SQ,[2] with the restart probability $c = 0.15$. As

[2] We do not include the query itself when retrieving the top-k results, same for the test queries on the Wiki data set.

Table 3. A Comparison of Inbound and Outbound Top-k Queries on Wiki ($\mathbf{c} = 0.15$ $k = 5$).

Query	Inbound Top-5 Results		Outbound Top-5 Results	
	Rank	Page Title	Rank	Page Title
Information Retrieval	1	IDF	1	Computer Science
	2	Index Term	2	Information Science
	3	Keyword (Internet Search)	3	Linguistics
	4	Precision and Recall	4	Association for Computing Machinery
	5	Recall (Information Retrieval)	5	Mathematics
Microsoft Office	1	Microsoft Excel	1	2006
	2	Microsoft Word	2	2007
	3	Microsoft Windows	3	2008
	4	Microsoft Office 2007	4	Microsoft
	5	Microsoft FrontPage	5	Microsoft Office 2007

shown, the results returned by the outbound top-k query are mostly some famous places in New York, such as the Metropolitan Museum of Art and the Radio City Music Hall. As such places are structural hubs in the graph, the random walk from the query node is thus very likely to reach them, making their outbound RWR scores high. In contrast, the results returned by the `Ink` query are less famous places but have strong correlations with the query place. For example, the top result for the query "Yankee Stadium" is Yankee Tavern, which is a local pub close to the stadium. Yankee fans may love to gather together at the pub to have some beer and talk about their favorite players.

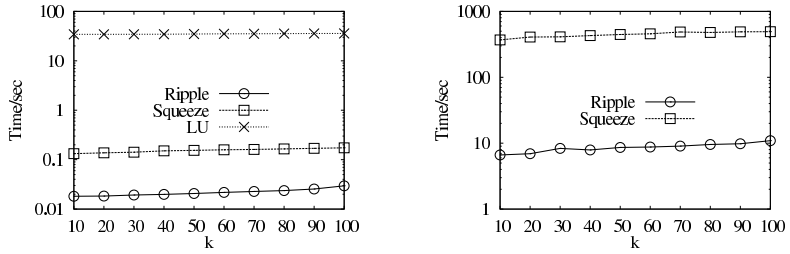
Table 3 shows the results for the queries "Information Retrieval" and "Microsoft Office" on Wiki. Again, we observe that the outbound top-k query tends to retrieve the pages that are popular, whereas the inbound top-k query obtains the pages that are more specific and strongly correlated to the query. For example, given the query "Information Retrieval", the results returned by `Ink` are all terminologies in the field of information retrieval, such as IDF and index term. In contrast, the outbound top-5 query returns general but more famous pages like Computer Science.

5.3 Efficiency Study

In this subsection, we study the efficiency of the proposed algorithms. An `Ink` query consists of two parameters: (1) the number k; and (2) the restart probability c. We set their default values as $k = 20$ and $c = 0.15$. We evaluate the effect of one parameter while the other is fixed at its default value, and run 1000 randomly generated queries with their average cost reported. LU and SQUEEZE are parameter-free, while RIPPLE has two parameters to tune: (1) s, the number of boundary nodes for expansion; and (2) t, the number of iterations after each expansion. We first fix $s = 30$ and $t = 1$ when comparing RIPPLE with the other two methods. Then we study the effect of s and t on the performance of RIPPLE.

Varying k. Figure 3 shows the running time of the three methods when k varies on 4SQ. As shown, k does not affect the running time of LU much, as the major cost of LU is the computation of the RWR scores of all the nodes in the graph. In contrast, the running time of SQUEEZE and RIPPLE increases with k, but at a quite slow rate. This phenomenon could be explained by the fact that, as k increases, the score gap between the k-th and the $(k + 1)$-th objects tends to become smaller. As a result, both SQUEEZE and RIPPLE need more iterations to retrieve the top-k results. Comparing the performance of the three methods, we find both SQUEEZE and RIPPLE outperform LU

significantly even though the pre-computation time of LU has already been excluded.[3] This fact suggests the branch-and-bound strategies used by SQUEEZE and RIPPLE are quite effective, they can largely prune the search space to avoid unnecessary RWR computations. Figure 3(b) shows the running time of SQUEEZE and RIPPLE on Wiki. We do not have the result of LU because the offline matrix decomposition stage fails to complete within one week on Wiki. Similarly, RIPPLE needs more iterations to produce the top-k results than SQUEEZE, but it takes much less time. Moreover, the performance gap between RIPPLE and SQUEEZE is even larger on Wiki (RIPPLE is faster by about two orders of magnitude) than on 4SQ. This is explained by the fact that RIPPLE is a local search algorithm and is not so sensitive to the data set size. Therefore, RIPPLE is suitable for extremely large graphs.



(a) Varying k on 4SQ. (b) Varying k on Wiki.

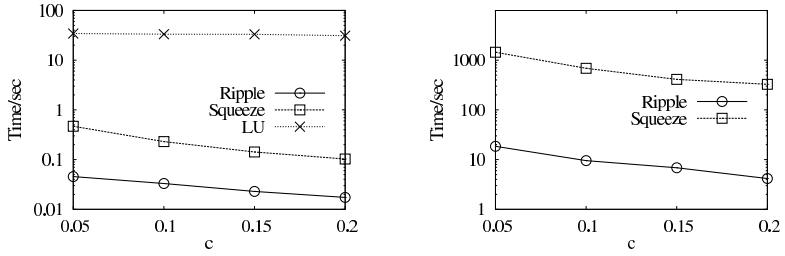
Fig. 3. Running time v.s. k.

Varying c. Figure 4 shows the running time of the three methods on 4SQ when c varies from 0.05 to 0.2. Again, SQUEEZE and RIPPLE significantly outperform LU under different values of c. Furthermore, it is worth mentioning that, for different values of c, LU needs to re-perform the LU matrix decomposition in the offline stage, while SQUEEZE and RIPPLE do not need any pre-computations.

The running time of SQUEEZE and RIPPLE decreases exponentially with c, which is in line with our expectation. As shown in Theorem 3, the error bound of SQUEEZE after i iterations is $(1-c)^i$. For a larger c, SQUEEZE needs much less iterations to produce the top-k results. Similarly, for RIPPLE, Theorem 5 and 6 suggest that the error bounds for both inside and outside nodes are much tighter under a larger c, thus the number of iterations are fewer. To understand this phenomenon from another perspective, RIPPLE leverages RWR locality to answer `Ink` queries. When c is large, the random surfer has a higher probability to jump back to the start node, thus the nodes close to the query node are more likely to appear in the correct top-k results, making the vicinity-based estimation prune the search space more effectively.

Effects of s and t. We proceed to study the effects of s and t on the performance of RIPPLE. Figure 5 shows the effect of s on RIPPLE on 4SQ and Wiki. As shown, on 4SQ, when s increases from 10 to 100, the running time of RIPPLE first decreases and then gradually becomes stable. The running time of RIPPLE on Wiki first decreases and then increases. This is explained by the fact that 4SQ (average degree is about 2.54) is sparser than Wiki (average degree is about 23.3). For the `Ink` queries on 4SQ, RIPPLE needs

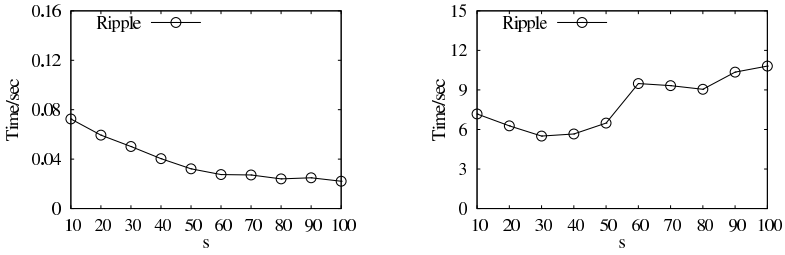
[3] For LU, the matrix decomposition phase takes about 87.5 hours to finish on the 4SQ graph.



(a) Varying c on 4SQ. (b) Varying c on Wiki.

Fig. 4. Running time v.s. c.

to expand more boundary nodes to ensure the vicinity is large enough to encompass the top-k nodes. In contrast, since Wiki is a denser graph, the required number of to-expand boundary nodes is expected to be smaller. If s is too large on Wiki, the cardinality of the vicinity grows too rapidly, which incurs unnecessary computational overhead. That said, the performance of RIPPLE is not quite sensitive to s on both data sets, which is a nice property in practice.



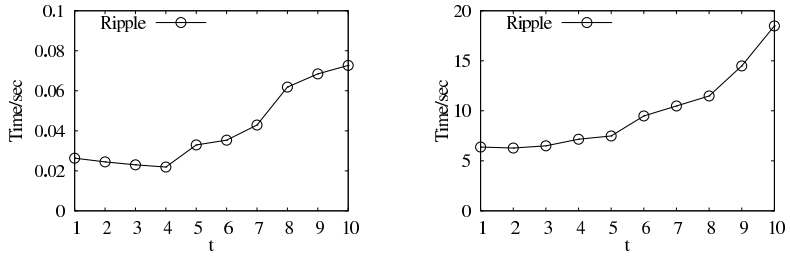
(a) Varying s on 4SQ. (b) Varying s on Wiki.

Fig. 5. Running time v.s. s for RIPPLE.

Figure 6 shows the performance of RIPPLE on 4SQ and Wiki when t increases from 1 to 10. We observe that the running time of RIPPLE is stable when t is small. However, the running time increases quite rapidly when t is too large. This suggests that iterating over a small vicinity for too many times cannot improve the efficiency of RIPPLE, but only incurs unnecessary computations. In practice, it is better to set t to a small value so that only a few iterations are performed before a new vicinity set is generated.

6 Related Work

The efficient computation of RWR has received a substantial amount of attention over the past decade. Though obtaining the closed-form solution of RWR requires the inversion of a matrix (Equation 1) and time-consuming, two popular strategies are widely adopted to address this problem: Monte Carlo sampling [2], [3] and power iteration [14]. Other techniques for efficiently approximating RWR have also been proposed. Tong *et al.* [15] introduced an efficient and novel algorithm for computing approximate RWR scores. Their method relies on a pre-processing step, which obtains the low-rank approximation of a large and sparse matrix. Zhu *et al.* [18] proposed to compute the



(a) Varying t on 4SQ. (b) Varying t on Wiki.

Fig. 6. Running time v.s. t for RIPPLE.

approximate PPR vector using the inverse P-distance [18]. The key idea is to partition all random walk tours into different layers according to their contributions, and given priority to those important layers when computing the PPR vector. Lofgren *et al.* [13] introduced the FAST-PPR algorithm based on the idea of bi-directional estimation. Given a threshold δ, FAST-PPR can compute the PPR from node s to node t, denoted as $\pi_s(t)$, in $O(\sqrt{d/\delta})$ time with a small error so long as $\pi_s(t) \geq \delta$. Unfortunately, these approximate algorithms cannot be easily applied to answer the `Ink` queries as it is hard to pre-specify the desired error bound for an ad-hoc query. Moreover, as suggested by RIPPLE, computing the RWR scores from all the nodes is actually unnecessary.

Fujiwara *et al.* [6] proposed an efficient approach for computing the exact RWR score between two nodes. As aforementioned, the key idea is to perform matrix LU decomposition as an offline step. Yu *et al.* [17] designed the IRWR algorithm for computing exact RWR when the graph in dynamically evolving. That is, IRWR focuses on how to obtain the exact RWR score between two nodes when some edges are inserted into or deleted from the graph.

Along another line, much attention has been paid to the outbound top-k search problem. The goal is to retrieve the k nodes with the highest RWR/PPR scores from a query node. Most of the existing techniques for answering outbound top-k search resort to the branch-and-bound strategy to prune the search space. Specifically, Gupta *et al.* [9] proposed the Basic Push Algorithm, which computes PPR bounds based on bookmark coloring. Bahmani *et al.* [4] proposed a Monte Carlo based method for finding approximate top-k neighbors. Their results demonstrate that, by precomputing and storing a number of short random walk tours for all the nodes in the graph, the top-k neighbors can be fast approximated with satisfactory accuracy. Fujiwara *et al.* [6] proposed the k-dash algorithm to identify the top-k nearest neighbors of a query node based on matrix LU decompostion. They later proposed an method [7] that does not rely on offline pre-computation, but estimates the lower and upper bounds in an on-line manner. However, the lower and upper bounds derived for the outbound top-k query cannot be easily adapted for our `Ink` query.

7 Conclusions

In this paper, we proposed the `Ink` query based on the random walk with restart measure. The `Ink` query retrieves the top-k nodes that have high weighted RWR scores

to a given query node. In order to efficiently process the `Ink` query in large graphs, we designed the SQUEEZE method and the RIPPLE method. SQUEEZE iteratively performs matrix vector multiplication and dynamically updates the lower and upper RWR bounds to generate the top-k result set. RIPPLE exploits RWR locality by maintaining a vicinity around the query node, and incrementally expands the vicinity to refine the RWR estimations. Our experimental results have demonstrated that both algorithms can answer `Ink` queries efficiently on large real-life graphs, while RIPPLE is especially suitable for extremely large graphs. Interesting future work includes investigating how the `Ink` query can benefit higher-level tasks such as link prediction, and how to adapt the RIPPLE method if the graph is stored in a distributed environment.

References

1. Andersen, R., Chung, F.R.K., Lang, K.J.: Local graph partitioning using pagerank vectors. In: FOCS. pp. 475–486 (2006)
2. Avrachenkov, K., Litvak, N., Nemirovsky, D., Smirnova, E., Sokol, M.: Quick detection of top-k personalized pagerank lists. In: WAW. pp. 50–61 (2011)
3. Bahmani, B., Chakrabarti, K., Xin, D.: Fast personalized pagerank on mapreduce. In: SIGMOD Conference. pp. 973–984 (2011), http://doi.acm.org/10.1145/1989323.1989425
4. Bahmani, B., Chowdhury, A., Goel, A.: Fast incremental and personalized pagerank. PVLDB 4(3), 173–184 (2010)
5. Fogaras, D., Rácz, B., Csalogány, K., Sarlós, T.: Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. Internet Mathematics 2(3), 333–358 (2005)
6. Fujiwara, Y., Nakatsuji, M., Onizuka, M., Kitsuregawa, M.: Fast and exact top-k search for random walk with restart. PVLDB 5(5), 442–453 (2012)
7. Fujiwara, Y., Nakatsuji, M., Shiokawa, H., Mishima, T., Onizuka, M.: Efficient ad-hoc search for personalized pagerank. In: SIGMOD Conference. pp. 445–456 (2013)
8. Fujiwara, Y., Nakatsuji, M., Yamamuro, T., Shiokawa, H., Onizuka, M.: Efficient personalized pagerank with accuracy assurance. In: KDD. pp. 15–23 (2012)
9. Gupta, M.S., Pathak, A., Chakrabarti, S.: Fast algorithms for topk personalized pagerank queries. In: WWW. pp. 1225–1226 (2008)
10. Jeh, G., Widom, J.: Scaling personalized web search. In: WWW. pp. 271–279 (2003)
11. Konstas, I., Stathopoulos, V., Jose, J.M.: On social networks and collaborative recommendation. In: SIGIR. pp. 195–202 (2009), http://doi.acm.org/10.1145/1571941.1571977
12. Liben-Nowell, D., Kleinberg, J.M.: The link prediction problem for social networks. In: CIKM. pp. 556–559 (2003)
13. Lofgren, P., Banerjee, S., Goel, A., Comandur, S.: FAST-PPR: scaling personalized pagerank estimation for large graphs. In: KDD. pp. 1436–1445 (2014), http://doi.acm.org/10.1145/2623330.2623745
14. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University (1998)
15. Tong, H., Faloutsos, C., Pan, J.Y.: Fast random walk with restart and its applications. In: ICDM. pp. 613–622 (2006)
16. Yu, A.W., Mamoulis, N., Su, H.: Reverse top-k search using random walk with restart. PVLDB 7(5), 401–412 (2014)
17. Yu, W., Lin, X.: Irwr: incremental random walk with restart. In: SIGIR. pp. 1017–1020 (2013)
18. Zhu, F., Fang, Y., Chang, K.C.C., Ying, J.: Incremental and accuracy-aware personalized pagerank through scheduled approximation. PVLDB 6(6), 481–492 (2013)