Reverse Top-k Search using Random Walk with Restart *

Adams Wei Yu[†§], Nikos Mamoulis[§], Hao Su[‡]
[†]School of Computer Science, Carnegie Mellon University
[§]Department of Computer Science, The University of Hong Kong
[‡]Computer Science Department, Stanford University
weiyu@cs.cmu.edu, nikos@cs.hku.hk, haosu@cs.stanford.edu

ABSTRACT

With the increasing popularity of social networks, large volumes of graph data are becoming available. Large graphs are also derived by structure extraction from relational, text, or scientific data (e.g., relational tuple networks, citation graphs, ontology networks, protein-protein interaction graphs). Node-to-node proximity is the key building block for many graph-based applications that search or analyze the data. Among various proximity measures, random walk with restart (RWR) is widely adopted because of its ability to consider the global structure of the whole network. Although RWR-based similarity search has been well studied before, there is no prior work on reverse top-k proximity search in graphs based on RWR. We discuss the applicability of this query and show that its direct evaluation using existing methods on RWR-based similarity search has very high computational and storage demands. To address this issue, we propose an indexing technique, paired with an on-line reverse top-k search algorithm. Our experiments show that our technique is efficient and has manageable storage requirements even when applied on very large graphs.

1. INTRODUCTION

Graph is a fundamental model for capturing the structure of data in a wide range of applications. Examples of real-life graphs include, social networks, the Web, transportation networks, citation graphs, ontology networks, and protein-protein interaction graphs. In most applications, a key concept is the node-to-node proximity, which captures the relevance between two nodes in a graph. A widely adopted proximity measure, due to its ability to capture the global structure of the graph, is *random walk with restart* (RWR). RWR proximity from node u to node v, is the probability for a random walk starting from u to reach v after infinite time; at any transition there is a chance α ($0 < \alpha < 1$) that the random walk restarts at u. Compared to other measures (like shortest path distance), a significant advantage of RWR is that it takes into account all the possible paths between two nodes. Other merits of RWR is that it can model the multi-faceted relationship between two nodes [13] and that RWR is *stable* to small changes in the graph [20]. RWR has been successfully applied in the search engine Google [21] to rank the importance of web pages. In addition, several other measures build upon RWR, including Personalized Pagerank [12], ObjectRank [5], Escape Probability [25], and PathSim [23].

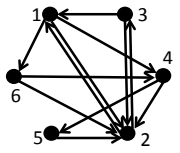
Many search and analysis tasks rely on proximity computations. These include, citation analysis in bibliographical graphs [14], link prediction in social networks [19], graph clustering [2], and making recommendations [16]. The *top-k RWR proximity* query retrieves the k nodes with the highest proximity from a given query node q in a graph. This problem has been investigated previously and efficient solutions have been proposed for it (e.g., [11, 3, 10]).

In this paper, we study the *reverse top-k RWR proximity* query: given a node q, find all the nodes that have q in their top-k RWR proximity sets. Reverse top-k queries can be used for detection of *spam* nodes in a graph. Search engines, such as Google, aggregate the RWR proximities from all other nodes to one node in a single value, known as PageRank. Thus, the proximity from web page u to v can be interpreted as the PageRank contribution that u makes to v. When a node q is suspected to be a spam web page, one could run a reverse top-k search on q, and find out the pages which give one of their top-k contributions to q. If the answer set contains a large proportion of web pages already labeled as spam, then q is likely to be a spam too. As another application, consider an author in a co-authorship network who wishes to find the set of people that regard himself as the one of their most important direct or indirect collaborators. The reverse top-k result can be used for identifying the likelihood of successful collaborations in the future. The size of an author's reverse top-k list is also an indicator of his popularity in the community. Finally, in a product co-purchase graph, a reverse top-k query of a product q can identify which products influence the buying of q. One can leverage this information to promote q in future transactions.

To the best of our knowledge, there is no previous work on reverse top-k RWR-based search in large graphs. In addition, extending solutions for top-k RWR search to compute reverse top-k queries is not trivial. Specifically, while for a top-k search we only need to find the top-k proximity set of *a single* node q, the reverse top-k search must compute the top-k sets of *all* nodes in the graph and check whether q appears in each of them. Therefore, a reverse top-k query is substantially more expensive than top-k RWR search. Figure 1 illustrates a toy graph of 6 nodes and the entire proximity matrix $\boldsymbol{P}$ computed from it; the i-th column $\boldsymbol{p}_i$ of $\boldsymbol{P}$ contains the proximity values *from* node i *to* all nodes in the graph (e.g., the proximity from node 1 to node 3 is 0.12). In each column $\boldsymbol{p}_i$, the $k = 2$ largest entries are shaded; these indicate the results of a top-2 query from node i (e.g., the top-2 query from node 3 returns nodes 2 and 3). Observe that for any given node q, to answer a top-

*Supported by grant HKU 715413E from Hong Kong RGC. Work done while the first author was with HKU.

Proceedings of the VLDB Endowment, Vol. 7, No. 5



$\boldsymbol{P}$	$\boldsymbol{p}_1$	$\boldsymbol{p}_2$	$\boldsymbol{p}_3$	$\boldsymbol{p}_4$	$\boldsymbol{p}_5$	$\boldsymbol{p}_6$
	0.32	0.24	0.24	0.19	0.20	0.18
	0.28	0.39	0.29	0.31	0.33	0.30
	0.12	0.17	0.27	0.13	0.14	0.13
	0.13	0.10	0.10	0.23	0.08	0.14
	0.06	0.04	0.04	0.10	0.18	0.06
	0.09	0.07	0.07	0.05	0.06	0.20

Figure 1: Example graph and its proximity matrix

2 query, we only have to compute and access the values of a single column, whereas a reverse top-2 query for a node i requires finding all the shaded entries in the i-th row (e.g., the reverse top-2 query for node 1 returns nodes 1, 2, and 5). To find whether an entry is shaded or not, we have to compute the entire matrix $\boldsymbol{P}$ and rank the values in each column. Computing the whole proximity matrix is both time and space-consuming, especially for large graphs.

We propose a reverse top-k query evaluation framework which alleviates this issue. In a nutshell, our approach computes (at a preprocessing step) from the graph G (having $|V|$ nodes) a *graph index*, which is based on a $K \times |V|$ matrix, containing in each column v the K largest approximate proximity values from v to any other nodes in G. K is application-dependent and represents the highest value of k in a practical query. At each column v of the index, the approximate values are lower bounds of the K largest proximity values from v to all other nodes, computed after adapting and partially executing Berkhin's *Bookmark Coloring Algorithm* (BCA) [7]. Given the graph index and a reverse top-k query q ($k \leq K$), we prove that the *exact* proximities from any node v to query q can be efficiently computed by applying the power method. By comparing these with the corresponding lower bounds taken from the k-th row of the graph index, we are able to determine which nodes (i.e., columns of $\boldsymbol{P}$) are certainly not in the reverse top-k result of q. For some of the remaining nodes, we may also be able to determine that they are certainly in the reverse top-k result, based on derived upper bounds for the k-th largest proximity value from them. Finally, for any candidate that remains, we progressively refine its approximate proximities, until based on its lower or upper bound we can determine if it should be in the result. The proximities refined during query processing can be updated into the graph index, making its values progressively more accurate for future queries.

Our contributions can be summarized as follows:

- We study for the first time reverse top-k proximity queries based on RWR in large graphs.
- We propose a dynamically refined, space-efficient index structure, which supports reverse top-k query evaluation. The index is paired with an efficient online query algorithm, which prunes a large number of nodes that are definitely in or not in the reverse top-k result and minimizes the required refinement for the remaining candidates.
- A side contribution of our online algorithm is a proof that we can apply the power method for computing the *exact* proximities *from* all nodes *to* a given node q. This result can serve as a module of any applications that need to compute RWR proximities *to* a given node.
- We conduct an experimental study demonstrating the efficiency of our framework, as well as the effectiveness of the reverse top-k RWR query in real graph applications.

The remainder of this paper is organized as follows. Section 2 provides a definition for the RWR-based proximity vector which captures the proximities from a given node to all other nodes and reviews methods for computing it. The reverse top-k RWR proximity search problem is formalized in Section 3 and the baseline brute force solution is discussed. In Section 4, we present our solution which is experimentally evaluated in Section 5. In Section 6, we briefly discuss previous work related to reverse top-k RWR proximity search. Finally, Section 7 concludes the paper.

General	
$\boldsymbol{A}$	Transition probability matrix
$\boldsymbol{P}$	Proximity matrix
$\boldsymbol{p}_u$	Proximity vector from node u to other nodes
$\boldsymbol{e}_u$	Unit vector having $\boldsymbol{e}_u(u) = 1$, and $\boldsymbol{e}_u(v) = 0\ \forall v \neq u$
p_u^{kmax}	The k-th largest entry of $\boldsymbol{p}_u$
BCA starting from node u	
$\boldsymbol{p}_u^t$ ($\boldsymbol{r}_u^t$)	Retained (residue) ink distribution at iteration t
$\boldsymbol{s}_u^t$ ($\boldsymbol{w}_u^t$)	Ink accumulated at hubs (non-hubs) at iteration t
$\hat{\boldsymbol{p}}_u$ ($\hat{\boldsymbol{p}}_u^t$)	Descending ranked list of $\boldsymbol{p}_u(\boldsymbol{p}_u^t)$
lb_u^t (ub_u^t)	Lower (upper) bound of $\hat{\boldsymbol{p}}_u^t(k)$

Table 1: Main notations

2. PRELIMINARIES

In this section, we first provide definitions for the RWR proximity matrix of a graph and the proximity vectors of nodes in it. Then, we review the Bookmark Coloring Algorithm (BCA) [7] for computing the RWR proximity from a given node to all other nodes, based on which our offline index is built. For a matrix $\boldsymbol{M}$, $\boldsymbol{m}_j$ or $\boldsymbol{m}_{*,j}$ denotes its j-th column, $\boldsymbol{m}_{i,*}$ denotes its i-th row, and $\boldsymbol{m}_{i,j}$ denotes the element of its i-th row and j-th column. For a vector $\boldsymbol{v}$, $\boldsymbol{v}(i)$ denotes its i-th entry and $\boldsymbol{v}(1:i)$ denotes its first i entries. Table 1 summarizes the main symbols used in the paper.

2.1 Definitions

Let $G = (V, E)$ be a directed graph, with a set $V = \{1, 2, ..., n\}$ of vertices and a set $E \subset V \times V$ of edges. Let $n = |V|, m = |E|$, $\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_n] \in \mathbb{R}^{n \times n}$ be the column-stochastic transition probability matrix, and $OD(i)$ be the out-degree of node i. We assume that $\boldsymbol{a}_{i,j} = \frac{1}{OD(j)}$ if edge $j \rightarrow i$ exists and $\boldsymbol{a}_{i,j} = 0$, otherwise.[1] In other words, the RWR transition probability from node j to any of its out-neighbors i only depends on the out-degree of j (i.e., all out-neighbors are equally likely to be visited). For a given node u, the RWR proximity values from it to other nodes is the solution of the following linear system w.r.t. $\boldsymbol{p}_u$:

$$\boldsymbol{p}_u = (1 - \alpha)\boldsymbol{A}\boldsymbol{p}_u + \alpha\boldsymbol{e}_u, \tag{1}$$

where $\boldsymbol{p}_u \in \mathbb{R}^n$ is the *proximity vector* of node u, with $\boldsymbol{p}_u(v)$ denoting the proximity from u to v; $\boldsymbol{e}_u \in \mathbb{R}^n$ is a unit vector having $\boldsymbol{e}_u(u) = 1$ and all other values set to 0, and $\alpha \in [0, 1]$ denotes the *restart probability* in RWR (typically, $\alpha = 0.15$). The analytical solution is

$$\boldsymbol{p}_u = \boldsymbol{P}\boldsymbol{e}_u, \tag{2}$$

where $\boldsymbol{P} = \alpha \cdot (\boldsymbol{I} - (1 - \alpha)\boldsymbol{A})^{-1} = [\boldsymbol{p}_1, \boldsymbol{p}_2, ..., \boldsymbol{p}_n]$ is called the *proximity matrix*. In fact, $\boldsymbol{P}$ can also be used to compute the Pagerank ($\boldsymbol{pr}$) and any Personalized Pagerank ($\boldsymbol{ppr}$) vector as follows:

$$\boldsymbol{pr} = \frac{1}{n}\boldsymbol{P}\boldsymbol{e}, \ \ \boldsymbol{ppr}_{\boldsymbol{v}} = \boldsymbol{P}\boldsymbol{v}, \tag{3}$$

[1] In the case where *dangling* nodes with no outgoing edges exist, we can simply delete them, or add a sink node which links to itself and is pointed by each dangling node.

where $\boldsymbol{e} \in \mathbb{R}^n$ has 1 in all entries and $\boldsymbol{v} \in \mathbb{R}^n$ is any given personalized vector such that $\forall 1 \leq i \leq n, v_i \geq 0$ and $\|\boldsymbol{v}\|_1 = \sum_{i=1}^{n} |v_i| = 1$.

Computing $\boldsymbol{P}$ at its entirety or partially is a key problem in different applications. Approaches like the iterative Power Method (PM) [21] and Monte Carlo Simulation (MCS) [9] can be used to compute an approximate value for a single proximity vector $\boldsymbol{p}_u$ and/or the entire matrix $\boldsymbol{P}$. PM converges to an accurate $\boldsymbol{p}_u$ while MCS is less accurate but faster. Next, we discuss in detail an efficient technique for deriving a lower-bound of $\boldsymbol{p}_u$.

2.2 Bookmark Coloring Algorithm (BCA)

Basic model. Berkhin [7] models RWR by a *bookmark coloring* process, which facilitates the efficient estimation of $\boldsymbol{p}_u$. We begin by injecting a unit amount of *colored ink* into u, with an α portion *retained* in u and the rest $(1-\alpha)$ portion evenly *distributed* to each of u's out-neighbors. Each node which receives ink retains an α portion of the ink and distributes the rest to its out-neighbors. At an intermediate step $t(= 0, 1, 2, ...)$, we can use two vectors $\boldsymbol{p}_u^t$, $\boldsymbol{r}_u^t \in \mathbb{R}^n$ to capture the ink distribution in the whole graph, where $\boldsymbol{p}_u^t(v)$ is the ink retained at node v, and $\boldsymbol{r}_u^t(v)$ is the *residue* ink to be distributed from v. When $\boldsymbol{r}_u^t(v)$ reaches 0 for all $v \in V$ (i.e., $\|\boldsymbol{r}_u\|_1 = 0$), $\boldsymbol{p}_u^t$ is exactly $\boldsymbol{p}_u$; the proximity vector $\boldsymbol{p}_u$ can be seen as a stable distribution of ink. In fact, BCA can stop early, at a time t, where $\boldsymbol{r}_u^t(v)$ values are small at all nodes v; $\boldsymbol{p}_u^t$ is then a sparse lower-bound approximation of $\boldsymbol{p}_u$ [7].

Hub effects. In the process of ink propagation, some of the nodes may have a high probability to receive new ink and distribute part of it again and again. Such nodes are called *hubs* and their set is denoted by $H = \{h_1, h_2, ..., h_{|H|}\}$. Without loss of generality, we assume that the first $|H|$ nodes in V are the hubs. If we knew how hubs distribute their ink across the graph (i.e., if we have precomputed the exact proximity vector $\boldsymbol{p}_h$ for each $h \in H$), we would not need to distribute their residue ink during the process of computing $\boldsymbol{p}_u$ for a node $u \in V \setminus H$. Instead, we could accumulate all the residue ink at hubs, and distribute it in batch at the end by a simple matrix multiplication. In [7], a greedy scheme is adopted to select hubs and implement this idea. It starts by applying BCA on one node and selecting the node with the largest retained ink as a hub. This process is repeated from another starting node to select another hub, until a sufficient number of hubs are chosen. Once the hub nodes are selected, we can use the power method (PM) to calculate the exact vector $\boldsymbol{p}_h$ for each $h \in \mathcal{H}$.

BCA using hubs. Assume that we have selected a set of hubs H and have pre-computed $\boldsymbol{p}_h$ for each $h \in \mathcal{H}$. To compute $\boldsymbol{p}_u$ for a non-hub node u, BCA [7] (and its revised version [2]) first injects a unit amount of ink to u, then u retains an α portion of the ink, and distributes the rest to u's out-neighbors. At each propagation step t, BCA picks a non-hub node v_t and distributes the residue ink $\boldsymbol{r}_u^{t-1}(v_t)$ to its out-neighbors. Two vectors $\boldsymbol{s}_u^t$ and $\boldsymbol{w}_u^t \in \mathbb{R}^n$ are introduced and maintained in this process. $\boldsymbol{s}_u^t$ is used to store the ink accumulated at hubs so far and $\boldsymbol{w}_u^t$ is used to store the ink retained at non-hub nodes. Thus, for a hub node h, $\boldsymbol{s}_u^t(h)$ is the ink accumulated at h by time t; this ink will be distributed to all nodes in batch after the final iteration, with the help of the (pre-computed) $\boldsymbol{p}_h$. For a non-hub node v, $\boldsymbol{w}_u^t(v)$ stores the ink retained so far at v (which will never be distributed). $\boldsymbol{w}_u^t(v)$ ($\boldsymbol{s}_u^t(v)$) is always zero for a hub (non-hub) node v. The following equations show how all vectors are updated at each step:

$$\boldsymbol{w}_u^t = \alpha \boldsymbol{r}_u^{t-1}(v_t) \cdot \boldsymbol{e}_{v_t} + \boldsymbol{w}_u^{t-1} \tag{4}$$

$$\boldsymbol{r}_u^t = (1-\alpha)\boldsymbol{r}_u^{t-1}(v_t) \cdot \boldsymbol{a}_{v_t} + [\boldsymbol{r}_u^{t-1} - \boldsymbol{r}_u^{t-1}(v_t) \cdot \boldsymbol{e}_{v_t}] \tag{5}$$

$$\boldsymbol{s}_u^t = \sum_{i \in H} \boldsymbol{r}_u^{t-1}(i) \cdot \boldsymbol{e}_i + \boldsymbol{s}_u^{t-1} \tag{6}$$

According to the first part of Eq. (4), an α ink portion of $\boldsymbol{r}_u^{t-1}(v_t)$ is retained at v_t. Eq. (5) subtracts the residue ink $\boldsymbol{r}_u^{t-1}(v_t)$ from v_t (second part) and evenly distributes the remaining $(1-\alpha)$ portion to v_t's out-neighbors (first part). Eq. (6) accumulates the ink that arrives at hub nodes. At any step t, BCA can compute $\boldsymbol{p}_u^t$ and use it to approximate $\boldsymbol{p}_u$, as follows:

$$\boldsymbol{p}_u^t = \boldsymbol{w}_u^t + \boldsymbol{P}_H \cdot \boldsymbol{s}_u^t \tag{7}$$

where $\boldsymbol{P}_H = [\boldsymbol{p}_1, \boldsymbol{p}_2, ..., \boldsymbol{p}_{|H|}, \boldsymbol{0}_{n \times (n-|H|)}] \in \mathbb{R}^{n \times n}$, i.e., $\boldsymbol{P}_H$ is the proximity matrix including only the (precomputed) proximity vectors of hub nodes and having 0's in all proximity entries of non-hub nodes. $\boldsymbol{p}_u^t$ is computed only when all residue values are small; in this case, it is deemed that $\boldsymbol{p}_u^t$ is a good approximation of $\boldsymbol{p}_u$. In order to reach this stage, at each step, a v_t with large residue ink should be selected. In [7], v_t is selected to be the node with the largest residue ink, while in [2] v_t is any node with more residue ink than a *propagation threshold* η. BCA terminates when the total residue ink does not exceed a *convergence threshold* ϵ or when there is no node with at least η residue ink.

3. PROBLEM FORMALIZATION

The reverse top-k RWR query is formally defined as follows:

PROBLEM 1. *Given a graph $G(V, E)$, a query node $q \in V$ and a positive integer k, find all nodes $u \in V$, for which $p_u^{kmax} \leq \boldsymbol{p}_u(q)$, where $\boldsymbol{p}_u$ is obtained by Eq.* (1) *and p_u^{kmax} is the k-th largest value in $\boldsymbol{p}_u$.*

A brute-force (BF) method for evaluating the query is to (i) compute the proximity vector $\boldsymbol{p}_u$ of every node u in the graph, and (ii) find all nodes u for which $p_u^{kmax} \leq \boldsymbol{p}_u(q)$. BF requires the computation of the entire *proximity matrix* $\boldsymbol{P}$. No matter which method is used to compute the exact $\boldsymbol{P}$ (e.g., PM or the state-of-the-art K-dash algorithm [10]), examining the top-k values at *each* column $\boldsymbol{p}_u$ results in a $O(n^3)$ total time complexity for BF (or $O(nm)$ for sparse graphs with n nodes and m edges), which is too high, especially for online queries on large-scale graphs.

There are several observations that guide us to the design of an efficient reverse top-k RWR algorithm. First, the expected number of nodes in the answer set of a reverse top-k query is k; thus there is potential of building an index, which can prune the majority of nodes definitely not in the answer set. Second, as noted in [4] and observed in our experiments, the power law distribution phenomenon applies on each proximity vector: typically, only few entries have significantly large and meaningful proximities, while the remaining values are tiny. Third, we observe that verifying whether the query node q lies in the top-k proximity set of a certain node u is a far easier problem than computing the *exact* top-k set of node u; we can efficiently derive upper and lower bounds for the proximities from u to all other nodes and use them for verification. In the next section, we introduce our approach, which achieves significantly better performance than BF.

4. OUR APPROACH

Our method focuses on two aspects: (i) avoiding the computations of unnecessary top-k proximity sets and (ii) terminating the computation of each top-k proximity set as early as possible. The overall framework contains two parts: an offline indexing module (Section 4.1) and an online querying algorithm (Section 4.2).

4.1 Offline Indexing

For our index design, we assume that the maximum k in any query does not exceed a predefined value K, i.e. $k \leq K$. For each node v, we compute proximity lower bounds to all other nodes and store the K largest bounds to a compact data structure. The index is relatively efficient to obtain, compared to computing the exact proximity matrix $\boldsymbol{P}$. Given a query q and a $k \leq K$, with the help of the index, we can prune nodes that are guaranteed not to have q in their top-k sets, thus avoiding a large number of unnecessary proximity vector computations. The index is stored in a compact format, so that it can fit in main memory even for large graphs. It also supports dynamic updating after a query has been completed; this way, its performance potentially improves for any future queries.

The lower bounds used in our index are based on the fact that, while running BCA from any node $u \in V$, each entry of $\boldsymbol{p}_u^t$ at iteration t is monotonically increasing w.r.t. t; formally:

PROPOSITION 1. $\forall u, v \in V$, $\boldsymbol{p}_u^1(v) \leq \boldsymbol{p}_u^2(v) \leq ... \leq \boldsymbol{p}_u(v)$.

PROOF. See [29]. □

Thus, after each iteration t of BCA from $u \in V$, we can have a lower bound $\boldsymbol{p}_u^t(v)$ of the real proximity value $\boldsymbol{p}_u(v)$ from u to any node $v \in V$. The following proposition shows that the k-th largest value in $\boldsymbol{p}_u^t$ serves as a lower bound for the k-th largest proximity value in $\boldsymbol{p}_u$:

PROPOSITION 2. *Let $\hat{\boldsymbol{p}}_u^t(k)$ be the k-th largest value in $\boldsymbol{p}_u^t$ after t iterations of BCA from u. Let $\hat{\boldsymbol{p}}_u(k)$ be the k-th largest value in $\boldsymbol{p}_u$. Then, $\hat{\boldsymbol{p}}_u^t(k) \leq \hat{\boldsymbol{p}}_u(k) = p_u^{kmax}$.*

PROOF. See [29]. □

Note that this is a nice property of BCA, which is not present in alternative proximity vector computation techniques (i.e., PM and MCS). Besides, we observe that by running BCA from a node u, the high proximity values stand out after only a few iterations. Thus, to construct the index, we run an adapted version of BCA from *each* node $u \in V$ that stops after a few iterations t to derive a lower-bound proximity vector $\boldsymbol{p}_u^t$. Only the K largest values of this vector are kept in descending order in a $\hat{\boldsymbol{p}}_u^t(1:K)$ vector. Our index consists of all these lower bounds; in Section 4.2, we explain how it can be used for query evaluation. In the remainder of this subsection, we provide details about our hub selection technique (Section 4.1.1), our adaptation of BCA for deriving the lower-bound proximity vectors and constructing the index (Section 4.1.2), and a compression technique that reduces the storage requirements of the index (Section 4.1.3).

4.1.1 Hub Selection

The hub selection method in [7], runs BCA itself to find hubs; its efficiency thus heavily relies on the graph size and the number of selected hubs. We use a simpler approach, which is independent of these factors and hence can be used for large-scale graphs. We claim that nodes with high in-degree or out-degree are already good candidates to be suitable hubs. Therefore we define H as the union of the sets of high in-degree nodes H_{in} and high out-degree nodes H_{out}. H_{in} (H_{out}) is the set of B nodes in V with the largest in-degree (out-degree). In Section 5, we investigate choices for parameter B.

4.1.2 BCA Adaptation

We propose an improved ink propagation strategy for BCA compared to those suggested by [7] and [2]. Instead of propagating a single node's residue ink at each iteration t, our strategy selects a subset of nodes L_t, which includes those having no less residue ink than a given *propagation threshold* η; i.e., $L_t = \{v \in V \setminus H | \boldsymbol{r}_u^{t-1}(v) \geq \eta\}$. η is selected such that only significant residue ink is propagated. The rules for updating $\boldsymbol{s}_u^t$ and $\boldsymbol{p}_u^t$ are the same as shown in Eq. (6) and (7), respectively. However, the updates $\boldsymbol{w}_u^t$ and $\boldsymbol{r}_u^t$ are performed as follows:

$$\boldsymbol{w}_u^t = \sum_{i \in L_t} \alpha \boldsymbol{r}_u^{t-1}(i) \cdot \boldsymbol{e}_i + \boldsymbol{w}_u^{t-1} \quad (8)$$

$$\boldsymbol{r}_u^t = \sum_{i \in L_t} (1-\alpha) \boldsymbol{r}_u^{t-1}(i) \cdot \boldsymbol{a}_i + [\boldsymbol{r}_u^{t-1} - \sum_{i \in L_t} \boldsymbol{r}_u^{t-1}(i) \cdot \boldsymbol{e}_i] \quad (9)$$

To understand the advantage of our strategy, note that the main cost of BCA at each iteration consists of two parts. The first is the time spent for selecting nodes to propagate ink from and the second is the time spent on updating vectors $\boldsymbol{r}_u^t$, $\boldsymbol{s}_u^t$, and $\boldsymbol{w}_u^t$.[2] Our approach reduces both costs. First, selecting a batch of nodes at a time significantly reduces the total remaining residue $\|\boldsymbol{r}_u^t\|_1$ in a single iteration and greatly reduces the overall number of iterations and thus the total number of vector updates. Second, since at each iteration both finding a single node or a set of nodes to propagate ink from requires a linear scan of $\boldsymbol{r}_u^{t-1}$, the total node selection time is also reduced.

Our BCA adaptation ends as soon as the total remaining ink $\|\boldsymbol{r}_u^t\|_1$ is no greater than a *residue threshold* δ. We observe that $\|\boldsymbol{r}_u^t\|_1$ drops drastically in the first few iterations of BCA and then slowly in the latter iterations. Thus, we select δ such that our BCA adaptation terminates only after a few iterations, deriving a rough approximation of $\boldsymbol{p}_u$ that is already sufficient to prune the majority of nodes during search.

The complete lower bound indexing procedure is described by Algorithm 1. Let t_u be the number of iterations until the termination of BCA from u and $\boldsymbol{t} = [t_1, t_2, ..., t_n]$. The index resulting from Algorithm 1 is denoted by $\mathcal{I}^{\boldsymbol{t}} = (\hat{\boldsymbol{P}}^{\boldsymbol{t}}, \boldsymbol{R}^{\boldsymbol{t}}, \boldsymbol{W}^{\boldsymbol{t}}, \boldsymbol{S}^{\boldsymbol{t}}, \boldsymbol{P}_H)$, where $\hat{\boldsymbol{P}}^{\boldsymbol{t}} = [\hat{\boldsymbol{p}}_1^{t_1}(1:K), ..., \hat{\boldsymbol{p}}_n^{t_n}(1:K)]$ is the *top-K lower bound matrix* storing the K largest values of each $\boldsymbol{p}_u^{t_u}$, $\boldsymbol{R}^{\boldsymbol{t}} = [\boldsymbol{r}_1^{t_1}, ..., \boldsymbol{r}_n^{t_n}]$ is the residue ink matrix, $\boldsymbol{W}^{\boldsymbol{t}} = [\boldsymbol{w}_1^{t_1}, ..., \boldsymbol{w}_n^{t_n}]$ is the non-hub retained ink matrix, $\boldsymbol{S}^{\boldsymbol{t}} = [\boldsymbol{s}_1^{t_1}, ..., \boldsymbol{s}_n^{t_n}]$ is the hub accumulated ink matrix and $\boldsymbol{P}_H$ is the hub proximity matrix. Whenever the context is clear, we simply denote $\boldsymbol{p}_u^{t_u}$ by $\boldsymbol{p}_u^t$ and the index by $\mathcal{I} = (\hat{\boldsymbol{P}}, \boldsymbol{R}, \boldsymbol{W}, \boldsymbol{S}, \boldsymbol{P}_H)$.

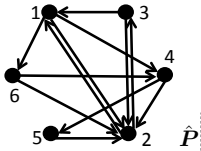
Algorithm 1 Lower Bound Indexing (LBI)

Input: Matrix $\boldsymbol{A}$, number K, Hubs H, Residue threshold δ, Propagation threshold η.
Output: Index $\mathcal{I} = (\hat{\boldsymbol{P}}, \boldsymbol{R}, \boldsymbol{W}, \boldsymbol{S}, \boldsymbol{P}_H)$.

```
for all h ∈ H do
    Compute p_h by power method or BCA;
for all nodes u ∈ V do
    t_u = 0; r_u^{t_u} = e_u; s_u^{t_u} = w_u^{t_u} = 0;
    while ||r_u^{t_u}||_1 > δ do
        t_u = t_u + 1;
        Update r_u^{t_u}, s_u^{t_u}, w_u^{t_u} by Eq. (9), (6), (8);
    Compute p_u^{t_u} by Eq. (7);
    p̂_u^{t_u} = top K entries of p_u^{t_u} in descending order;
```

Figure 2 illustrates the result of our indexing approach on the toy graph of Figure 1, for $\alpha = 0.15$. First, by setting $B = 1$, we select the two nodes with the highest in- and out-degrees to become hubs. These are nodes 1 and 2. For these two nodes the exact proximity vectors $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are computed and stored in the

[2]Recall that $\boldsymbol{p}_u^t$ needs not be updated at each iteration and is only computed at the end of BCA or when an approximation of $\boldsymbol{p}_u$ should be obtained.



$\boldsymbol{p}_1$	$\boldsymbol{p}_2$	$\boldsymbol{p}_3^{t_3}$	$\boldsymbol{p}_4^{t_4}$	$\boldsymbol{p}_5^{t_5}$	$\boldsymbol{p}_6^{t_6}$
0.32	0.24	0.24	0.10	0.20	0.10
0.28	0.39	0.29	0.17	0.33	0.17
0.12	0.17	0.27	0.07	0.14	0.07
0.13	0.10	0.10	0.19	0.08	0.10
0.06	0.04	0.04	0.08	0.18	0.02
0.09	0.07	0.07	0.03	0.06	0.18

$\hat{\boldsymbol{P}}$

$\hat{\boldsymbol{p}}_1$	$\hat{\boldsymbol{p}}_2$	$\hat{\boldsymbol{p}}_3^{t_3}$	$\hat{\boldsymbol{p}}_4^{t_4}$	$\hat{\boldsymbol{p}}_5^{t_5}$	$\hat{\boldsymbol{p}}_6^{t_6}$
0.32	0.39	0.29	0.19	0.33	0.18
0.28	0.24	0.27	0.17	0.20	0.17
0.13	0.17	0.24	0.10	0.18	0.10

Figure 2: Example of top-3 lower bound index

hub proximity matrix $\boldsymbol{P}_H = [\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}]$. For the remaining nodes, we run our BCA adaptation with propagation threshold $\eta = 10^{-4}$ and residue threshold $\delta = 0.8$, which results in the $\boldsymbol{p}_3^{t_3}$–$\boldsymbol{p}_6^{t_6}$ vectors shown in the figure. Finally, we select from each of $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3^{t_3}, \ldots, \boldsymbol{p}_6^{t_6}\}$ the top-K values (for $K = 3$) and create the lower bound matrix $\hat{\boldsymbol{P}} = [\hat{\boldsymbol{p}}_1, \hat{\boldsymbol{p}}_2, \hat{\boldsymbol{p}}_3^{t_3}, \hat{\boldsymbol{p}}_4^{t_4}, \hat{\boldsymbol{p}}_5^{t_5}, \hat{\boldsymbol{p}}_6^{t_6}]$, as shown in the figure. Note that $\|\boldsymbol{r}_3^{t_3}\|_1 = \|\boldsymbol{r}_5^{t_5}\|_1 = 0$ and $\|\boldsymbol{r}_4^{t_4}\|_1 = \|\boldsymbol{r}_6^{t_6}\|_1 = 0.36$.

4.1.3 Compact Storage of the Index

The space complexity for the hub proximity matrix $\boldsymbol{P}_H$ of $\mathcal{I}$ is $O(|H|n)$, where $|H|$ (n) is the number of hub (total) nodes. The matrix may not fit in memory if n and $|H|$ are large. We apply a compression technique for $\boldsymbol{P}_H$, based on the observation that the values of a proximity vector follow a power law distribution; in each vector $\boldsymbol{p}_h \in \boldsymbol{P}_H$, the great majority of values are tiny; only a small percentage of these values are significantly large. Therefore, we perform rounding by zeroing all values lower than a given *rounding threshold* ω. In our implementation, we choose an ω that can save much space without losing reverse top-k search precision. If sufficient hubs are selected, matrices $\boldsymbol{R}, \boldsymbol{W}, \boldsymbol{S}$ are sparse, so the storage cost for the index $\mathcal{I}$ will mainly be due to $\hat{\boldsymbol{P}}$ and the rounded $\boldsymbol{P}_H$. The following theorem gives an estimation for the total index storage requirements after the rounding operation.

THEOREM 1. *$\forall h \in H$, given rounding threshold ω, if the values of $\boldsymbol{p}_h$ follow a power law distribution, i.e., the sorted value $\hat{\boldsymbol{p}}_h(i) \propto i^{-\beta}$, where $0 < \beta < 1$ is the exponent parameter, then the space required to store the whole index is $O(Kn + (1-\beta)^{\frac{1}{\beta}}|H|\omega^{-\frac{1}{\beta}}n^{1-\frac{1}{\beta}})$.*

PROOF. Let $\hat{\boldsymbol{p}}_h(i) = \gamma i^{-\beta}$. As

$$1 = \sum_{i=1}^{n} \hat{\boldsymbol{p}}_h(i) = \gamma \sum_{i=1}^{n} i^{-\beta} \approx \gamma n^{\beta-1} \int_0^1 x^{-\beta} \mathrm{d}x = \frac{\gamma n^{\beta-1}}{1-\beta}$$

we have $\gamma \approx (1-\beta)n^{\beta-1}$ and $\hat{\boldsymbol{p}}_h(i) \approx (1-\beta)n^{\beta-1}i^{-\beta}$. Let $\hat{\boldsymbol{p}}_h(l^*) \geq \omega$, then we have

$$l^* \leq (1-\beta)^{\frac{1}{\beta}}\omega^{-\frac{1}{\beta}}n^{1-\frac{1}{\beta}}$$

Since only less than l^* entries need to be stored for a single hub node, we need $(1-\beta)^{\frac{1}{\beta}}|H|\omega^{-\frac{1}{\beta}}n^{1-\frac{1}{\beta}}$ space for $\boldsymbol{P}_H$. Plus the top-K lower bound space requirement $O(Kn)$, the total index storage would be $O(Kn + (1-\beta)^{\frac{1}{\beta}}|H|\omega^{-\frac{1}{\beta}}n^{1-\frac{1}{\beta}})$. □

Let $\underline{\boldsymbol{p}}_u^{t_u}$ be the approximated proximities constructed by Eq. (7) with rounded hub proximities $\underline{\boldsymbol{P}}_H$. We can trivially show that Propositions 1 and 2 hold for $\underline{\boldsymbol{p}}_u^{t_u}$. Thus, $\underline{\boldsymbol{p}}_u^{t_u}$ is still an increasing lower bound of $\boldsymbol{p}_u$ and $\underline{\boldsymbol{p}}_u^{t_u}$ can replace the $\boldsymbol{p}_u^{t_u}$ in our index. In the following, we give a bound for the error caused by rounding.

PROPOSITION 3. *Given rounding threshold ω and $\hat{\boldsymbol{p}}_h(i) \approx \gamma i^{-\beta}$, where $\gamma = (1-\beta)n^{\beta-1}$, then for $\forall u \in V$,*

$$\|\boldsymbol{p}_u^{t_u} - \underline{\boldsymbol{p}}_u^{t_u}\|_1 \leq 1 - (\frac{1-\beta}{\omega n})^{\frac{1}{\beta}-1}.$$

PROOF. See [29]. □

We empirically observed (see Section 5) that our rounding approach can save huge amounts of space and the real storage requirements are even much smaller than the theoretical bound given by Theorem 1. Meanwhile, the actual error is much smaller than the theoretical bound by Proposition 3, and more importantly, it has minimal effect to the reverse top-k results. To keep the index notation uncluttered, we use $\boldsymbol{P}_H$ to also denote the rounded hub proximities (i.e., $\underline{\boldsymbol{P}}_H$) and $\boldsymbol{p}_u^{t_u}$ to denote the corresponding rounded proximity vectors $\underline{\boldsymbol{p}}_u^{t_u}$ computed using $\underline{\boldsymbol{P}}_H$.

4.2 Online Query Algorithm

This section introduces our online reverse top-k search technique. Given a query node $q \in V$, we perform search in two steps. First, we compute the exact proximity from each $u \in V$ to q using a novel and efficient method (Section 4.2.1). In the second step (Section 4.2.2), for each node u we use the index described in Section 4.1 to prune u or add u to the search result, by deriving a lower and an upper bound (denoted as lb_u^t and ub_u^t) of u's k-th largest proximity value $\boldsymbol{p}_u^{kmax}$ to other nodes and comparing it with its proximity to q. For nodes u that cannot be pruned or confirmed as results, we refine lb_u^t and ub_u^t using our index incrementally, until u is pruned or becomes a confirmed result. The refinement is used to update the index for faster future query processing (Section 4.2.3). In this section, we use $\boldsymbol{p}_u$ and $\boldsymbol{p}_{*,u}$ interchangeably to denote the u-th column of the proximity matrix $\boldsymbol{P}$; also note that $lb_u^t = \hat{\boldsymbol{p}}_u^t(k)$ and ub_u^t is the upper bound of $\hat{\boldsymbol{p}}_u(k)$ $(= \boldsymbol{p}_u^{kmax})$ w.r.t. to $\hat{\boldsymbol{p}}_u^t$.

4.2.1 RWR Proximity to the Query Node

The first step of our method is to compute the *exact* proximities from *all* other nodes to the query q. Although a lot of previous work has focused on computing the proximities from a given node u to all other nodes (i.e., a column $\boldsymbol{p}_{*,u}$ of the proximity matrix $\boldsymbol{P}$), there have only been a few efforts on how to find the proximities from all nodes to a node q (i.e., a row $\boldsymbol{p}_{q,*}$ of $\boldsymbol{P}$). The authors of the SpamRank algorithm [6] suggest computing approximate proximity vectors $\boldsymbol{p}_{*,u}$ for all $u \in V$, and taking all $\boldsymbol{p}_{q,u}$ to form $\boldsymbol{p}_{q,*}$. However, to get an exact result, which is our target, such a method would require the computation of the entire $\boldsymbol{P}$ to a very high precision, which leads to unacceptably high cost. A heuristic algorithm is proposed in [8], which first selects the nodes with high probability to be the large proximity contributors to the query node, and then computes their proximity vectors. This method requires the computation of several proximity vectors $\boldsymbol{p}_{*,u}$ to find only a subset of entries in $\boldsymbol{p}_{q,*}$. [1] introduces a local search algorithm that examines only a small fraction of nodes, deriving, however, only an approximation of $\boldsymbol{p}_{q,*}$.

Although it seems inevitable to compute the whole matrix $\boldsymbol{P}$ to get the exact proximities from all nodes to q, we show that this problem can be solved by the power method and has the same complexity as calculating a single column of $\boldsymbol{P}$. Our result is novel and constitutes an important contribution not only for the reverse top-k search problem that we study in this paper, but also for any problem that includes finding the proximities from all nodes to a given node.

For example, our method could be used as a module in SpamRank [6] to find PageRank contributions that all nodes make to a given web page q precisely and efficiently.

First of all, we note that $\boldsymbol{p}_{q,*}$ is essentially the q-th row of $\boldsymbol{P}$, hence, $\boldsymbol{p}_{q,*} = \boldsymbol{e}_q^T \boldsymbol{P} = \alpha \boldsymbol{e}_q^T \cdot (\boldsymbol{I} - (1-\alpha)\boldsymbol{A})^{-1}$, or equivalently

$$\boldsymbol{p}_{q,*}^T = (1-\alpha)\boldsymbol{A}^T \boldsymbol{p}_{q,*}^T + \alpha \boldsymbol{e}_q$$

(see Section 2 for the definitions of $\boldsymbol{A}$, $\boldsymbol{e}_q$, and $\boldsymbol{e}$). An interesting observation is that $\boldsymbol{p}_{*,u}$ and $\boldsymbol{p}_{q,*}$ are actually the solutions of the following linear systems respectively,

$$\boldsymbol{x}_u = (1-\alpha)\boldsymbol{A}\boldsymbol{x}_u + \alpha \boldsymbol{e}_u \tag{10}$$

$$\boldsymbol{x}_q = (1-\alpha)\boldsymbol{A}^T \boldsymbol{x}_q + \alpha \boldsymbol{e}_q \tag{11}$$

which share the same structure except, that either $\boldsymbol{A}$ or $\boldsymbol{A}^T$ is used as the coefficient matrix. This similarity motivates us to apply the power method. Just as Eq. (10) can be solved by the iterative power method on matrix $[(1-\alpha)\boldsymbol{A} + \alpha \boldsymbol{e}_u \boldsymbol{e}^T]$:

$$\boldsymbol{x}_u^{i+1} = (1-\alpha)\boldsymbol{A}\boldsymbol{x}_u^i + \alpha \boldsymbol{e}_u = [(1-\alpha)\boldsymbol{A} + \alpha \boldsymbol{e}_u \boldsymbol{e}^T]\boldsymbol{x}_u^i, \tag{12}$$

we hope that the linear system (11) could be solved by the following iterative method:

$$\boldsymbol{x}_q^{i+1} = (1-\alpha)\boldsymbol{A}^T \boldsymbol{x}_q^i + \alpha \boldsymbol{e}_q. \tag{13}$$

However, showing that the sequence generated by Eq. (13) can successfully converge to the solution of Eq. (11) is not trivial, as the proof of the convergence of Eq. (12) does not apply for Eq. (13). The main difference between the two is as follows. In Eq. (12), if $\|\boldsymbol{x}_u^0\|_1 = 1$, then $\|\boldsymbol{x}_u^i\|_1 = \boldsymbol{e}^T \boldsymbol{x}_u^i = 1$ for $i = 1, 2, ...$ Hence we can have the r.h.s. of Eq. (12) to prove $\{\boldsymbol{x}_u^i\}_i$ to be a power method's series and thus converges. Conversely, the sequence $\{\boldsymbol{x}_q^i\}_i$ is not non-expansive in the general case and we may have $\|\boldsymbol{x}_q^{i+1}\|_1 > \|\boldsymbol{x}_q^i\|_1$. In other words, we cannot transform Eq. (13) to the form of the r.h.s. of Eq. (12) to prove $\{\boldsymbol{x}_q^i\}_i$ to be a power method's series, so there is no obvious guarantee that it will converge. We therefore have to prove that Eq. (13) converges to a unique vector, which is the solution of Eq. (11). Fortunately, using techniques very different from the original convergence proof of Eq. (12), we show that Eq. (13) indeed converges to a unique solution, from an arbitrary initialization.

Let us lift $\boldsymbol{x}_q \in \mathbb{R}^n$ to space $\mathbb{R}^{n+1}$ by introducing $\boldsymbol{z}_q = \begin{bmatrix} \boldsymbol{x}_q \\ 1 \end{bmatrix}$. The affine Equation (11) is now equivalent to

$$\boldsymbol{z}_q = \boldsymbol{D}_q \boldsymbol{z}_q \tag{14}$$

where $\boldsymbol{D}_q = \begin{bmatrix} (1-\alpha)\boldsymbol{A}^T & \alpha \boldsymbol{e}_q \\ \boldsymbol{0}_{1\times n} & 1 \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)}$. Then the first n columns of (14) is exactly (11). Note that $\boldsymbol{z}_q$ is an eigenvector of $\boldsymbol{D}_q$ corresponding to eigenvalue 1. We will prove that $\boldsymbol{z}_q$ is in fact the dominant eigenvector, therefore System (14) can be solved by the power method.

THEOREM 2. *Let λ_1 and λ_2 be the first two largest eigenvalues of $\boldsymbol{D}_q$. Let $\boldsymbol{z}_q^0 = [(\boldsymbol{x}_q^0)^T, 1]^T, \boldsymbol{z}_q^* = [\boldsymbol{p}_{q,*}, 1]^T \in \mathbb{R}^{(n+1)}$, where $\boldsymbol{x}_q^0$ is any vector in $\mathbb{R}^n$, and let*

$$\boldsymbol{z}_q^{i+1} = \boldsymbol{D}_q \boldsymbol{z}_q^i = \boldsymbol{D}_q^{i+1} \boldsymbol{z}_q^0 \tag{15}$$

then the following conclusions hold:

(a) $\lambda_1 = 1$ with multiplicity 1, and $\lim_{i\to\infty} \boldsymbol{z}_q^i = \boldsymbol{z}_q^*$, $\lim_{i\to\infty} \boldsymbol{x}_q^i = \boldsymbol{p}_{q,*}$.

(b) $\lambda_2 = 1-\alpha$; the convergence rate of (15) and (13) is $1-\alpha$;

(c) For convergence tolerance ϵ, if $i > \log \frac{\epsilon}{\alpha} / \log(1-\alpha)$, then $\|\boldsymbol{z}_q^{i+1} - \boldsymbol{z}_q^i\|_1 \equiv \|\boldsymbol{x}_q^{i+1} - \boldsymbol{x}_q^i\|_1 < \epsilon$.

PROOF. **(a)** Note that the row sum of $\boldsymbol{D}_q$ cannot exceed 1. In fact, for $\alpha > 0$, it is obvious that the q-th row and the last row have row sum 1 and all other rows have row sum $1-\alpha < 1$. So the spectral radius $\rho(\boldsymbol{D}_q) \le \max_i \sum_j (\boldsymbol{D}_q)_{ij} \le 1$. On the other hand, $\boldsymbol{z}_q^* \neq \boldsymbol{0}$ satisfies Eq. (14), which implies that $\boldsymbol{z}_q^*$ is the eigenvector of $\boldsymbol{D}_q$ with eigenvalue 1. Thus, $\lambda_1 = \rho(\boldsymbol{D}_q) = 1$.

Note that any eigenvector of value 1 must be a fixed point of Eq. (14). Therefore, if we can show that the sequence $\{\boldsymbol{z}_q^i\}_i$ converges to a nonzero point, it must be the unique eigenvector, and then the multiplicity of λ_1 is 1. In the following, we will prove that this statement is true. It is easy to verify that

$$\boldsymbol{D}_q^i = \begin{bmatrix} (1-\alpha)^i (\boldsymbol{A}^T)^i & \alpha \sum_{j=0}^{i-1} (1-\alpha)^j (\boldsymbol{A}^T)^j \boldsymbol{e}_q \\ \boldsymbol{0}_{1\times n} & 1 \end{bmatrix}$$

Since $\|\boldsymbol{A}^T\| = \rho(\boldsymbol{A}^T) = 1$, it follows that

$$\|(1-\alpha)^i (\boldsymbol{A}^T)^i\| \le (1-\alpha)^i \|\boldsymbol{A}^T\|^i \le (1-\alpha)^i, \text{so}$$

$$\lim_{i\to\infty} \boldsymbol{D}_q^i = \begin{bmatrix} \boldsymbol{0}_{n\times n} & \alpha[\boldsymbol{I} - (1-\alpha)\boldsymbol{A}^T)]^{-1} \boldsymbol{e}_q \\ \boldsymbol{0}_{1\times n} & 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{P}^T \boldsymbol{e}_q \\ \boldsymbol{0} & 1 \end{bmatrix},$$

implying that

$$\lim_{i\to\infty} \boldsymbol{z}_q^i = \lim_{i\to\infty} \boldsymbol{D}_q^i \boldsymbol{z}_q^0 = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{P}^T \boldsymbol{e}_q \\ \boldsymbol{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_q^0 \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{p}_{q,*}^T \\ 1 \end{bmatrix},$$

where $\boldsymbol{p}_{q,*}^T = \boldsymbol{P}^T \boldsymbol{e}_q$. Hence $\lim_{i\to\infty} \boldsymbol{z}_q^i = \boldsymbol{z}_q^*$ and $\lim_{i\to\infty} \boldsymbol{x}_q^i = \boldsymbol{p}_{q,*}$. This also certifies that there is a unique convergence point of (15), so the multiplicity of λ_1 is 1.

(b) Rewrite $\boldsymbol{D}_q = (1-\alpha) \begin{bmatrix} \boldsymbol{A}^T & \boldsymbol{0}_{n\times 1} \\ \boldsymbol{0}_{1\times n} & 1 \end{bmatrix} + \alpha \boldsymbol{F}_q$, where $\boldsymbol{F}_q = \begin{bmatrix} \boldsymbol{0}_{n\times n} & \boldsymbol{e}_q \\ \boldsymbol{0}_{n\times 1} & 1 \end{bmatrix}$. Let $\boldsymbol{\xi} = \begin{bmatrix} \boldsymbol{0}_{1\times n} \\ 1 \end{bmatrix}$. It is easy to verify that $\boldsymbol{D}_q^T \boldsymbol{\xi} = \boldsymbol{\xi}$. As $\rho(\boldsymbol{D}_q^T) = \rho(\boldsymbol{D}_q) = 1$, $\boldsymbol{\xi}$ is the eigenvector corresponding to the largest eigenvalue of $\boldsymbol{D}_q^T$ and it is unique, since $\boldsymbol{D}_q$ and $\boldsymbol{D}_q^T$ has the same eigenvalue multiplicity. Now we leverage the following lemma to assist the rest of proof.

LEMMA 1. *(From page 4 of [27]) If $\boldsymbol{\xi}_i$ is an eigenvector of $\boldsymbol{A}$ corresponding to the eigenvalue λ_i, $\boldsymbol{\zeta}_j$ is an eigenvector of $\boldsymbol{A}^T$ corresponding to λ_j and $\lambda_i \neq \lambda_j$, then $\boldsymbol{\xi}_i^T \boldsymbol{\zeta}_j = 0$.*

By Lemma 1, the second largest eigenvector $\boldsymbol{\zeta}$ of $\boldsymbol{D}_q$ must be orthogonal to $\boldsymbol{\xi}$, i.e., $\boldsymbol{\zeta}^T \boldsymbol{\xi} = 0$. By the structure of $\boldsymbol{\xi}$, it must be true that $\boldsymbol{\zeta} = \begin{bmatrix} \boldsymbol{\mu} \\ 0 \end{bmatrix}$, $\boldsymbol{\mu}$ is some vector in $\mathbb{R}^n$, which implies $\boldsymbol{F}_q \boldsymbol{\zeta} = \boldsymbol{0}$. Hence, $\boldsymbol{D}_q \boldsymbol{\zeta} = (1-\alpha) \begin{bmatrix} \boldsymbol{A}^T & \boldsymbol{0} \\ \boldsymbol{0} & 1 \end{bmatrix} \boldsymbol{\zeta} = (1-\alpha) \begin{bmatrix} \boldsymbol{A}^T \boldsymbol{\mu} \\ 0 \end{bmatrix}$. As $\boldsymbol{D}_q \boldsymbol{\zeta} = \lambda_2 \boldsymbol{\zeta}$, we have $\boldsymbol{A}^T \boldsymbol{\mu} = \frac{\lambda_2}{1-\alpha} \boldsymbol{\mu}$, indicating that $\boldsymbol{\mu}$ is an eigenvector of $\boldsymbol{A}^T$. Since $\boldsymbol{A}$ is a transition matrix, $\frac{\lambda_2}{1-\alpha} \le \rho(\boldsymbol{A}) = 1$, so $\lambda_2 \le 1-\alpha$. It is easy to verify that for $\boldsymbol{\zeta} = \begin{bmatrix} \boldsymbol{e}_{n\times 1} \\ 0 \end{bmatrix}$, $\boldsymbol{D}_q \boldsymbol{\zeta} = (1-\alpha)\boldsymbol{\zeta}$, so $\lambda_2 = 1-\alpha$. In addition, the convergence rate of (15) is dictated by $|\lambda_2|/|\lambda_1| = 1-\alpha$.

(c) Since $\{\boldsymbol{z}_q^i\}_i$ is the power method's series of $\boldsymbol{D}_q$, we have $\|\boldsymbol{z}_q^{i+1} - \boldsymbol{z}_q^i\|_1 \approx \|(\frac{|\lambda_2|}{|\lambda_1|})^i (1 - \frac{|\lambda_2|}{|\lambda_1|})\| = (1-\alpha)^i \alpha$. Hence, $i > \log \frac{\epsilon}{\alpha} / \log(1-\alpha)$ can lead to $\|\boldsymbol{z}_q^{i+1} - \boldsymbol{z}_q^i\|_1 < \epsilon$. □

Theorem 2 shows that sequence $\{\boldsymbol{x}_q^i\}_i$, computed by Eq. (13) indeed converges and also gives the estimated number of iterations.

Since it is part of the power method series $\{\boldsymbol{z}_q^i\}_i$, we can call Eq. (13) a power method; Algorithm 2 illustrates how to use it in solving System (11) and deriving $\boldsymbol{p}_{q,*}$. Note that the algorithm terminates as soon as the series converges based on the *convergence threshold* ϵ (line 6).As it takes $O(m)$ operations in each iteration (where $m = |E|$ is the number of edges), the time complexity of the algorithm is $O\left(\frac{\log \frac{\epsilon}{\alpha}}{\log(1-\alpha)} \cdot m\right)$.

Algorithm 2 Power Method for Proximity to Node (PMPN)

Input: Matrix $\boldsymbol{A}$, Query q, Convergence tolerance ϵ.
Output: Proximities $\boldsymbol{p}_{q,*}$ from all nodes to q.
1: Initialize $\boldsymbol{x}_q^0$ as any vector $\in \mathbb{R}^n$;
2: $i = 0$;
3: **repeat**
4: $\quad \boldsymbol{x}_q^{i+1} = (1-\alpha)\boldsymbol{A}^T\boldsymbol{x}_q^i + \alpha\boldsymbol{e}_q$;
5: $\quad i = i + 1$;
6: **until** $\|\boldsymbol{x}_q^i - \boldsymbol{x}_q^{i-1}\| < \epsilon$ ▷ convergence of PMPN
7: $\boldsymbol{p}_{q,*} = (\boldsymbol{x}_q^i)^T$;

4.2.2 Upper Bound for the k-largest Proximity

After having computed $\boldsymbol{p}_{q,*}$, we know for each $u \in V$, the exact proximity $\boldsymbol{p}_u(q)(= \boldsymbol{p}_{q,u})$ from u to q. Now, we access the k-th row of the lower bound matrix $\hat{\boldsymbol{P}}$ of the index (see Section 4.1) and prune all nodes u for which $lb_u^t = \hat{\boldsymbol{p}}_u^t(k) > \boldsymbol{p}_u(q)$. Obviously, if the k-th largest lower bound from u to any other node exceeds $\boldsymbol{p}_u(q)$, then it is not possible for q to be in the set of k closest nodes to u. For each node u that is not pruned, we compute an upper bound ub_u^t for the k-th largest proximity from u to any other node, using the information that we have about u in the index. If $\boldsymbol{p}_u(q) \geq ub_u^t$, then u is definitely in the answer set of the reverse top-k query. Otherwise, node u needs further processing.

We now show how to compute ub_u^t for a node u. Note that from the index, we have the descending top-K lower bound list $\hat{\boldsymbol{p}}_u^t$ and the residue ink vector $\boldsymbol{r}_u^t$. For $j = 1, 2, ..., k-1$, let

$$\Delta_j^t = \hat{\boldsymbol{p}}_u^t(j) - \hat{\boldsymbol{p}}_u^t(j+1) \tag{16}$$

$$z_0^t = 0, \text{and } z_j^t = z_{j-1}^t + j \cdot \Delta_{k-j}^t, 1 \leq j \leq k-1 \tag{17}$$

Then,

$$ub_u^t = \begin{cases} \hat{\boldsymbol{p}}_u^t(k-j) - \frac{z_j^t - \|\boldsymbol{r}_u^t\|_1}{j}, & \text{if } \exists\, j \in [1, k-1], \\ & \text{s.t. } z_{j-1}^t < \|\boldsymbol{r}_u^t\|_1 \leq z_j^t \\ \hat{\boldsymbol{p}}_u^t(1) + \frac{\|\boldsymbol{r}_u^t\|_1 - z_{k-1}^t}{k}, & \text{if } \|\boldsymbol{r}_u^t\|_1 > z_{k-1}^t \end{cases} \tag{18}$$

Figures 3 and 4 illustrate the intuition and the derivation of ub_u^t. Assume that $k = 5$ and the first k values of $\hat{\boldsymbol{p}}_u^t$ are as shown on the left of the figures, while the total remaining ink $\|\boldsymbol{r}_u^t\|_1$ is shown on the right of the figures. The best possible case for the k-th value of $\boldsymbol{p}_u$ is when $\|\boldsymbol{r}_u^t\|_1$ is distributed such that (i) only the first k values may receive some ink, while all others receive zero ink and (ii) the ink is distributed in a way that maximizes the updated k-th value. To achieve (ii), $\hat{\boldsymbol{p}}_u^t$ could be viewed as a staircase the k highest steps of which are fit tightly in a container. If we pour the total residue ink $\|\boldsymbol{r}_u^t\|_1$ into the container, the level of the ink will correspond to the value of ub_u^t. Δ_j^t is the difference between j-th and $(j+1)$-th step of the staircase, while z_j^t is the ink required to pour in order for its level in the container to reach the $(k-j)$-th step. The first line of Eq. (18) corresponds to the case illustrated by Figure 3, where ub_u^t is smaller than $\hat{\boldsymbol{p}}_u(1)$, while the example of Figure 4 corresponds to the case of the second line, where the whole staircase is covered by residue ink ($\|\boldsymbol{r}_u^t\|_1 > z_{k-1}^t$).

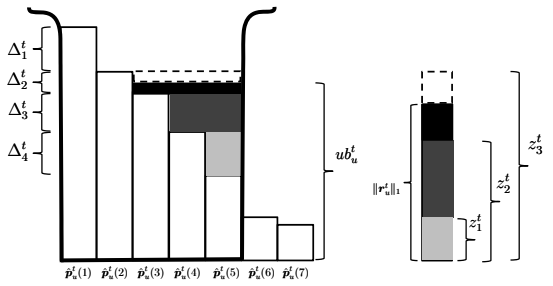


Figure 3: Upper bound, $k = 5, z_2^t < \|\boldsymbol{r}_u^t\|_1 \leq z_3^t$

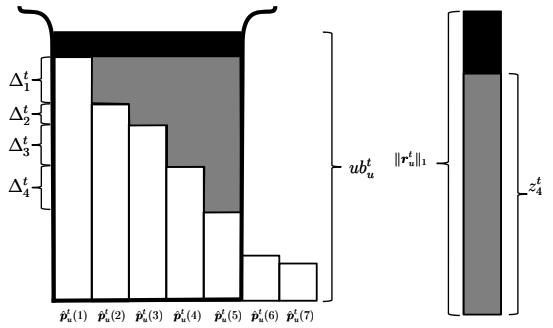


Figure 4: Upper bound, $k = 5, \|\boldsymbol{r}_u^t\|_1 > z_4^t$

The following proposition states that ub_u^t is indeed an upper bound of the real k-largest value p_u^{kmax} and is monotonically decreasing as $\boldsymbol{p}_u^t(\hat{\boldsymbol{p}}_u^t)$ is refined by later iterations.

PROPOSITION 4. *$\forall u \in V$, $ub_u^1 \geq ub_u^2 \geq ... \geq p_u^{kmax}$.*

PROOF. See [29]. □

Algorithm 3 is a procedure for deriving the upper bound ub_u^t, given $\hat{\boldsymbol{p}}_u$, $\|\boldsymbol{r}_u^t\|_1$, and k. The algorithm simulates pouring $\|\boldsymbol{r}_u^t\|_1$ into the container by gradually computing the z_j^t values for $j = 1, 2, \ldots, k-1$, until $z_{j-1}^t < \|\boldsymbol{r}_u^t\|_1 \leq z_j^t$, which indicates that the residue ink $\|\boldsymbol{r}_u^t\|_1$ can level up to $\hat{\boldsymbol{p}}_u^t(k-j)$. If $\|\boldsymbol{r}_u^t\|_1 > z_{k-1}^t$, the whole staircase is covered and the algorithm computes ub_u^t by the second line of Eq. (18). The complexity of Algorithm 3 is $O(k)$, which is quite low compared to other modules.

Algorithm 3 Upper Bound Computation (UBC)

Input: Matrix $\boldsymbol{A}$, Number k, Node u, Lower bound vector $\hat{\boldsymbol{p}}_u^t$, Residue ink vector $\boldsymbol{r}_u^t$.
Output: Upper bound ub_u^t of the k-th largest proximity from u.
1: $z_0^t = 0$;
2: **for** $j = 1$ **to** $k-1$ **do**
3: $\quad$ Compute Δ_{k-j}^t by Eq. (16);
4: $\quad$ Compute z_j^t by Eq. (17);
5: $\quad$ **if** $z_{j-1}^t < \|\boldsymbol{r}_u^t\|_1 \leq z_j^t$ **then**
6: $\quad\quad$ Compute ub_u^t by first line of Eq. (18);
7: $\quad\quad$ **return** ub_u^t;
8: Compute and **return** ub_u^t by second line of Eq. (18);

4.2.3 Candidate Refinement and Index Update

When $\hat{\boldsymbol{p}}_u^t(k) \leq \boldsymbol{p}_{q,u} < ub_u^t$, we cannot be sure whether u is a reverse top-k result or not and we need to further refine the bounds

$\hat{\boldsymbol{p}}_u^t(k)$ and ub_u^t. First, we apply one step of BCA in continuing the computation of $\boldsymbol{p}_u^t$ and update $\hat{\boldsymbol{p}}_u^t$ (lines 6-7 of Algorithm 1). Then, we apply Algorithm 3 to compute a new ub_u^t. This step-wise refinement process is repeated while $\hat{\boldsymbol{p}}_u^t(k) \leq \boldsymbol{p}_{q,u} < ub_u^t$; it stops once (i) $\boldsymbol{p}_{q,u} < \hat{\boldsymbol{p}}_u^t(k)$, which means that q is not contained in the top-k list of u, or (ii) $\boldsymbol{p}_{q,u} \geq ub_u^t$, which means that u definitely has q as one of its top-k nearest nodes. In our empirical study, we observed that for most of the candidates u, the process terminates much earlier before the lower and upper bounds approach the exact value $\boldsymbol{p}_{q,u}$. Thus, many computations are saved.

If, due to a reverse top-k search, $\boldsymbol{p}_u^t(\hat{\boldsymbol{p}}_u^t)$ has been updated, we dynamically update the index to include this change. In addition, we update the corresponding stored values for $\boldsymbol{r}_u^t$, $\boldsymbol{s}_u^t$, and $\boldsymbol{w}_u^t$. Due to this update, future queries will use tighter lower and upper bounds for u.

The complete online query (OQ) method is summarized by Algorithm 4. After computing the exact proximities to q (line 1), the algorithm examines all $u \in V$ and while a node u is a candidate based on the lower bound $\hat{\boldsymbol{p}}_u^{t_u}(k)$ (line 4), we first check (line 5) whether the lower bound is the actual proximity (this happens when $\|\boldsymbol{r}_u^{t_u}\|_1 = 0$); in this case, u is added to the result set C and the loop breaks. Otherwise, the upper bound $ub_u^{t_u}$ is computed (line 8) to verify whether u can be confirmed to be a result; if u is not a result (line 13), lines 6-7 of Algorithm 1 are run to *refine* $\hat{\boldsymbol{p}}_u^{t_u}(k)$; after the update, the lower bound condition is re-checked to see whether u can be pruned or another loop is necessary. Note that the update besides increasing the values of $\hat{\boldsymbol{p}}_u^{t_u}(k)$ (i.e., increasing the chances for pruning), it also reduces $ub_u^{t_u}$, therefore the revised upper bound $ub_u^{t_u}$ may render u a query result.

Algorithm 4 Online Query (OQ)

Input: Matrix $\boldsymbol{A}$, Query q, Number k, Index $\mathcal{I}$.
Output: Reverse top-k Set C of q, Updated Index $\mathcal{I}$.

1: Compute the exact proximities $\boldsymbol{p}_{q,*}$ by Algorithm 2;
2: Initialize $C = \emptyset$;
3: **for all** $u \in V$ **do**
4: **while** $\boldsymbol{p}_{u,q} \geq \hat{\boldsymbol{p}}_u^{t_u}(k)$ **do**
5: **if** $\|\boldsymbol{r}_u^{t_u}\|_1 = 0$ **then**
6: $C = C \cup u$; ▷ $\hat{\boldsymbol{p}}_u^{t_u}(k) = \hat{\boldsymbol{p}}_u(k)$, so u is a result
7: break;
8: Compute $ub_u^{t_u}$ by Algorithm 3;
9: **if** $\boldsymbol{p}_{u,q} \geq ub_u^{t_u}$ **then**
10: $C = C \cup u$; ▷ u becomes a result
11: break;
12: **else**
13: Update $\hat{\boldsymbol{p}}_u^{t_u}(k)$ by Algorithm 1;
14: Save the updated $\hat{\boldsymbol{P}}, \boldsymbol{R}, \boldsymbol{W}, \boldsymbol{S}$ to $\mathcal{I}$;

We now illustrate OQ with our running example. Consider the graph and the constructed index shown in Figure 2. Assume that $q = 1$ (i.e., the query node is node 1 in the graph) and $k = 2$. The first step is to compute $\boldsymbol{p}_{q,*}$ using Algorithm 2; the result is $\boldsymbol{p}_{q,*} = [0.32, 0.24, 0.24, 0.19, 0.20, 0.18]$. Now OQ loops through all nodes u and checks whether $\boldsymbol{p}_{u,q} \geq \hat{\boldsymbol{p}}_u^{t_u}(k)$. For the first node $u = 1$, we have $0.32 > 0.28$ and $\hat{\boldsymbol{p}}_u^{t_u}(k)$ is the actual proximity $\boldsymbol{p}_u(k)$ (recall that node 1 is a hub in our example, whose proximities to other nodes have been computed), thus 1 is a result. The same holds for $u = 2$ ($0.24 \geq 0.24$ and node 2 is a hub). For $u = 3$, observe that $\boldsymbol{p}_{u,q} < \hat{\boldsymbol{p}}_u^{t_u}(k)$ (i.e., $0.24 < 0.27$); therefore node 3 is safely pruned (i.e., OQ does not enter the while loop for $u = 3$). Node $u = 4$ satisfies $\boldsymbol{p}_{u,q} \geq \hat{\boldsymbol{p}}_u^{t_u}(k)$ ($0.19 > 0.17$) and $\|\boldsymbol{r}_u^{t_u}\|_1 > 0$, therefore the upper bound $ub_u^{t_u} = 0.36$ is computed by Algorithm 3, however, $\boldsymbol{p}_{u,q} < ub_u^{t_u}$, therefore we are still uncertain whether node 4 is a reverse top-k result. A loop of Algorithm 1 is run to update $\hat{\boldsymbol{p}}_u^{t_u}(k)$ to 0.23 (line 13); now node 4 is pruned because $\boldsymbol{p}_{u,q} < \hat{\boldsymbol{p}}_u^{t_u}(k)$ ($0.19 < 0.23$). Continuing the example, node 5 is immediately added to the result since $\boldsymbol{p}_{5,q} = \hat{\boldsymbol{p}}_5^{t_5}(k)$ and $\|\boldsymbol{r}_5^{t_5}\|_1 = 0$, whereas node 6 is pruned after the refinement of $\boldsymbol{p}_6^{t_6}$.

The following theorem shows the time complexity of OQ.

THEOREM 3. *The time complexity of OQ in worst case is*

$$O\left(\left(\frac{\log\frac{\epsilon}{\alpha} + |Cand| \cdot \log\frac{\eta}{\delta}}{\log(1-\alpha)}\right) \cdot m\right)$$

where the ϵ is the convergence threshold of Algorithm 2, δ is the residue threshold and η is the propagation threshold of Algorithm 1, $Cand$ is the set of candidates that could not be pruned immediately by the index and $m = |E|$ is the number of graph edges.

PROOF. The cost of a query includes the cost of Algorithm 2, which is $O\left(\frac{\log\frac{\epsilon}{\alpha}}{\log(1-\alpha)} \cdot m\right)$, as discussed in Section 4.2.1, and the cost of examining and refining the candidates (lines 2 to 14 of OQ). The worst case is that all nodes in $Cand$ cannot be pruned or confirmed as result until we compute their exact k-th largest proximity values by repeating line 7 of Algorithm 1, i.e., until the maximum residue ink $\max_i\{\boldsymbol{r}_u^{t_u}(i)\}$ at any node drops below η. Within an iteration, the update of one node u requires at most $O(m)$ operations Besides, each iteration is expected to shrink $\max_i\{\boldsymbol{r}_u^{t_u}(i)\}$ by a factor around $(1-\alpha)$. Recall that since $\max_i\{\boldsymbol{r}_u^{t_u}(i)\} \leq \delta$, the total number of iterations τ required to terminate BCA by making it smaller than η satisfies $\max_i\{\boldsymbol{r}_u^{t_u}(i)\} \cdot (1-\alpha)^\tau \approx \eta$, i.e., $\tau \approx \frac{\log\frac{\eta}{\max_i\{\boldsymbol{r}_u^{t_u}(i)\}}}{\log(1-\alpha)} \leq \frac{\log\frac{\eta}{\delta}}{\log(1-\alpha)}$. Therefore, the total time complexity in the worst case is $O\left(\left(\frac{\log\frac{\epsilon}{\alpha} + |Cand| \cdot \log\frac{\eta}{\delta}}{\log(1-\alpha)}\right) \cdot m\right)$. □

As we show in Section 5.3, in practice, $|Cand|$ is extremely small compared to n and most of the candidates can be pruned or confirmed within significantly fewer than τ iterations. Hence, the empirical performance of OQ is far better than the worst case.

5. EXPERIMENTAL EVALUATION

This section experimentally evaluates our approach , which is implemented in Matlab 2012b. Our testbed is a cluster of 500 AMD Opteron cores (800MHz per core) with a total of 1TB RAM. Since our indexing algorithm can be fully parallelized (i.e., the approximate proximity vectors of nodes are computed independently), we evenly distributed the workload to 100 cores to implement the indexing task. Each online query was executed at a single core and the memory used at runtime corresponds to the size of our index (i.e., at most a few GBs as reported in Table 2). Hence, our solution can also run on a commodity machine.

5.1 Datasets

We conducted our efficiency experiments on a set of unlabeled graphs. The number $n = |V|$ ($m = |E|$) of nodes (edges) of each graph are shown in Table 2. Web-stanford-cs[3] and Web-stanford[4] were crawled from stanford.edu. Each node is a web domain and a directed link stands for a hyperlink between two nodes. Epinions[4] is a 'who-trust-whom' social network from a consumer review site epinions.com; each node is a member of this site, and a directed edge $i \rightarrow j$ means that member i trusts j. Web-google[4] a web graph collected by Google. Experiments on two additional datasets are included in [29].

[3] law.di.unimi.it/datasets.php
[4] snap.stanford.edu/data/

5.2 Index Construction

We first evaluate the cost for constructing our index (Section 4.1) and its storage requirements for different graphs and sizes $|H|$ of hub sets. After tuning, we set the index construction parameters (see Section 4.1) as follows: propagation threshold $\eta = 10^{-4}$, residue threshold $\delta = 0.1$, hub vector rounding threshold $\omega = 10^{-6}$ for the first three graphs, and $\omega = 5 \cdot 10^{-6}$ for the largest one. In all cases, K=200, the convergence threshold $\epsilon = 10^{-10}$ and the restart parameter $\alpha = 0.15$. For a study on the effect of the different parameters and the rationale of choosing their default values, see [29].

Table 2 shows the index construction time for different graphs, for various values of the hub selection parameter B, which result in different sizes $|H|$ of hub sets. The last column shows the time to compute the entire proximity matrix $\boldsymbol{P}$ and its size on disk, which represents the brute-force (BF) approach of just pre-computing and using $\boldsymbol{P}$ for reverse top-k queries. The value in parentheses in the last column is the minimum possible cost for our index, derived by just storing the top-K lower bound matrix $\hat{\boldsymbol{P}}$ and disregarding the storage of the hub proximities $\boldsymbol{P}_H$ and matrices $\boldsymbol{R}$, $\boldsymbol{W}$, and $\boldsymbol{S}$. The last three rows for each graph show the space that our index would have if we had not applied the compression technique discussed in Section 4.1.3, the actual space of our index, and the predicted space according to our analysis in Section 4.1.3 (i.e., using Theorem 1 with $\beta = 0.76$, as indicated by [4]). The reported times sum up the running time at each core, assuming the worst case of having just one single-core machine. Note that the actual time is roughly the reported time divided by the number of cores (100).

We observe that the best number of hubs to select in terms of both index construction cost and index size on disk depends on the graph sparsity. For Web-stanford-cs, which is sparse graph, it suffices to select less than 1% of the nodes with the highest in- and out- degrees as hubs, while for the denser Epinions and Web-stanford graphs $1\% - 2\%$ of the nodes should be selected. The index construction is much faster than the entire $\boldsymbol{P}$ computation, especially for larger and sparser graphs (e.g., for Web-google it takes as little as 1.8% of the time to construct $\boldsymbol{P}$). The time is not affected too much by the number of selected hubs.

The same observation also holds for the size of our index, which is much smaller than the entire $\boldsymbol{P}$ and a few times larger than the baseline storage of the top-K lower bound matrix $\hat{\boldsymbol{P}}$. Although our index also stores the hub matrix $\boldsymbol{P}_H$ and matrices $\boldsymbol{R}$, $\boldsymbol{W}$, and $\boldsymbol{S}$, its space is reasonable; the index can easily be accommodated in the main memory of modern commodity hardware. The predicted space according to our analysis is in most cases an overestimation, due to an under-estimation of the power law effect on the sparsity of proximity matrices. Note that our rounding approach generally achieves significant space savings especially on large graphs (e.g., Web-google). For each dataset, the index that we are using in subsequent experiments is marked in bold.

Web-stanford-cs ($\|V\| = 9914$, $\|E\| = 36854$)					
B	**50**	100	200	300	
$\|H\|$	**82**	175	355	530	
time (s)	**31.5**	31.6	34.2	40.4	365.5
no rounding (MB)	**55.2**	57.4	65.3	77.9	
actual space (MB)	**39.6**	41.8	49.7	62.4	786 (15.8)
pred. space (MB)	**44.7**	93.5	188	280	
Epinions ($\|V\| = 75879$, $\|E\| = 508837$)					
B	1000	1500	**2000**	3000	
$\|H\|$	1484	2101	**2690**	3853	
time (s)	15827	12285	**11565**	10792	139860
no rounding (MB)	2778	2309	**2284**	2721	
actual space (MB)	2310	1696	**1538**	1716	46071 (121)
pred. space (MB)	4220	5924	**7551**	10763	
Web-stanford ($\|V\| = 281903$, $\|E\| = 2312497$)					
B	1000	1500	**2000**	3000	
$\|H\|$	1932	2866	**3804**	5586	
time (s)	85503	89196	**97462**	111200	3263500
no rounding (MB)	6506	8237	**10209**	14069	
actual space (MB)	1907	1639	**1595**	1638	635754 (451)
pred. space (MB)	3977	5681	**7393**	10645	
Web-google ($\|V\| = 875713$, $\|E\| = 5105039$)					
B	5000	**10000**	20000	50000	
$\|H\|$	9598	**18871**	37148	86246	
time (s)	1024200	**1107400**	2206300	2865300	60162000
no rounding (MB)	73362	**137113**	264315	607615	
actual space (MB)	5387	**4727**	4888	6897	6718720 (1466)
pred. space (MB)	2874	**4298**	7103	14639	

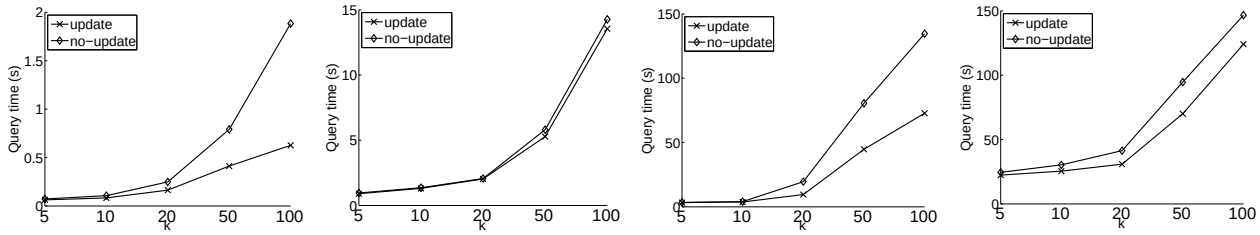
Table 2: Index construction time and space cost

5.3 Online Query Performance

We now evaluate the performance of our index and our on-line reverse top-k algorithm. We run a sequence of 500 queries on the indexes created in Section 5.2 and report average statistics for them.

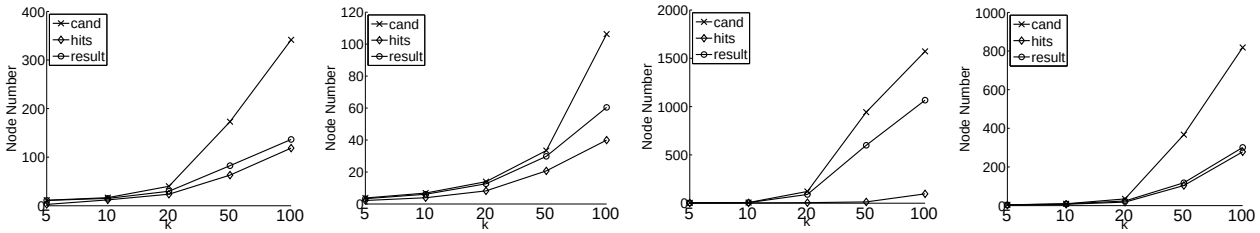
Query Efficiency. Figure 5 shows the average runtime cost of reverse top-k queries on different graphs, for different values of k and with different options for using the index. Series "update" denotes that after each query is evaluated, the index is updated to "save" the changes in the $\hat{\boldsymbol{P}}$, $\boldsymbol{R}$, $\boldsymbol{W}$, and $\boldsymbol{S}$ matrices, while "no-update" means that the original index is used for each of the 500 queries. We separated these cases in order to evaluate whether our index update policy brings benefits to subsequent queries which apply on a more refined index. The case of "update" also bears the cost of updating the corresponding matrices. In either case, query evaluation is very fast compared to the brute-force approach of computing the entire $\boldsymbol{P}$ (the time needed for this is already reported in the last column of Table 2) for each graph. The update policy results in significant reduction of the average query time in small and dense graphs; however, for larger and sparser graphs the index update has marginal effect in the query time improvement because there is a higher chance that subsequent queries are less dependent on the index refinement done by previous ones. Note that the workload includes 500 queries, which is a small number compared to the size of the graphs; we expect that for larger workloads the difference will be amplified on large graphs.

Pruning Power of Bounds. Figure 6 shows, for the same queries and the "update" case only, the average number (per query) of the candidates that are not immediately filtered using the lower bounds of the index and also the number of nodes from these candidates that are immediately identified as *hits* (i.e., results) after their upper bound computation. This means that only (*candidates*$-$*hits*) nodes (i.e., columns of $\hat{\boldsymbol{P}}$) need to be refined on average for each query. We also show the average number of actual results for each experimental setting. The plots show that the number of candidates are in the order of k and a significant percentage of them are immediately identified as results (based on their upper bounds) without needing refinement, a fact that explains the efficiency of our approach. In addition, the cost required for the refinement of these candidates is much lower compared to the cost for computing their exact proximity vectors. For example, computing the exact proximity vector $\boldsymbol{p}_u$ for a node u in Web-google takes more than 65 seconds, while our method requires just 0.15 seconds to refine a candidate in a reverse top-100 query on the same graph, on average. Another observation is that in some graphs, like Web-stanford-cs and Web-google, the *hits* number is very close to the *results* number. This suggests that when the accuracy demand is not high, an approximated query algorithm, which only takes the *hits* as result and stops further exploration, would save even more time.



(a) Web-stanford-cs (b) Epinions (c) Web-stanford (d) Web-google

Figure 5: Search performance on different graphs, varying k



(a) Web-stanford-cs (b) Epinions (c) Web-stanford (d) Web-google

Figure 6: Number of candidates and immediate hits on different graphs, varying k

Effectiveness of Index Refinement. Figure 7 shows the cost of individual reverse top-100 queries in the 500-query workload on the Web-stanford graph, with and without the index update option. Obviously, some queries are harder than others, depending on the number of candidates that should be refined and the refinement cost for them. We observe an increase in the gap between the query costs as the query ID increases, which is due to the fact that as the index gets updated the next queries in the sequence are likely to take advantage of the update to avoid redundant refinements (which would have to be performed if the index was not updated). For these queries that take advantage of the updates (i.e., the ones toward the end of the sequence), the cost is much lower compared to the case where they are run against a non-updated index. In the following, all experiments refer to the "update" case, i.e., the index is updated after each query evaluation.

Cumulative Cost. Figure 8 compares the cumulative cost of a workload that includes all nodes from the Web-stanford-cs graph as queries with the cumulative cost of two versions of the BF method on the same workload (k=10). The *infeasible* BF method (IBF) first constructs the exact $\boldsymbol{P}$ matrix, keeps the exact top-K proximity values for each node u, and then evaluates each reverse top-k query q at the minimal cost of accessing the q-th row of $\boldsymbol{P}$ and the k-th proximity value for each $u \in V$. However, since IBF requires materializing in memory the whole $\boldsymbol{P}$ (e.g., 6.7TB for Web-google), it becomes infeasible for large graphs. An alternative, *feasible* BF (FBF) method computes the entire $\boldsymbol{P}$, but keeps in memory only the *exact* top-K proximities of each node. Then, at query evaluation, FBF uses our approach in Section 4.2.1 to compute the exact RWR proximities to the query node from each node in the graph and then uses the exact pre-computed proximities to verify the reverse top-k results. As the figure shows, IBF has a high initial cost for computing $\boldsymbol{P}$ and afterward the cost for each query is very low. FBF bears the same overhead as IBF to compute $\boldsymbol{P}$, but requires longer query time. Our approach has little initial overhead of constructing our index and thereafter a modest cost for evaluating each query and updating the index. From the figure, we can see that the cumulative cost of our method is always lower than that of FBF and lower than IBF at the first 60% queries. (We emphasize again that IBF is infeasible for large graphs.) Besides, in practice, reverse top-k search is only applied on a small percentage of nodes (e.g., less than 10%); thus, its cumulative cost is low even when compared to that of IBF. In summary, the overhead of computing $\boldsymbol{P}$ in both versions of BF is very high, especially for large graphs, given the fact that not too many reverse top-k queries are issued, in practice.

Rounding Effect. We also tested the effect of using of the rounded hub proximity matrix $\underline{\boldsymbol{P}}_H$ in our index instead of the exact hub proximity matrix $\boldsymbol{P}_H$ on the query results (see Section 4.1.3). We used the 500 query workload on the Web-stanford-cs graph and for each query, we recorded the Jaccard similarity $\frac{|R_1 \cup R_2|}{|R_1 \cap R_2|}$ between the exact query results R_1 when using $\boldsymbol{P}_H$ and the results R_2 when using $\underline{\boldsymbol{P}}_H$ (i.e., our compressed index). Figure 9 plots the average similarity between the results of the same query when using $\boldsymbol{P}_H$ or $\underline{\boldsymbol{P}}_H$, for different values of k and the ω rounding threshold. Observe that for $\omega = 10^{-5}$ or smaller (as adopted in our setting), the results obtained with $\underline{\boldsymbol{P}}_H$ for different k are exactly the same as those obtained with $\boldsymbol{P}_H$. Even a larger threshold $\omega = 10^{-4}$ achieves an average precision of around 99% for all the tested k values. Thus, the rounding technique (Section 4.1.3) loses almost no accuracy, while saving a lot of space, as indicated by the results of Table 2.

5.4 Search Effectiveness

The experiments of this section demonstrate the effectiveness of reverse RWR top-k search in some real graph-based applications.

Spam detection. Webspam[5] is a web host graph containing 11402 web hosts, out of which, 8123 are manually labeled as "normal", 2113 are "spam", and the remaining ones are "undecided". There are 730774 directed edges in the graph. We verify the use of reverse RWR top-k search on spam detection by applying reverse top-5 search on all the spam and normal nodes, and check what

[5]barcelona.research.yahoo.net/webspam/datasets/uk2006/

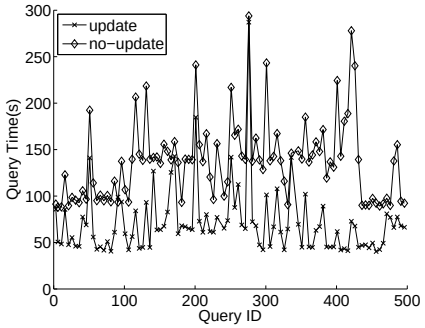


Figure 7: Cost of individual queries

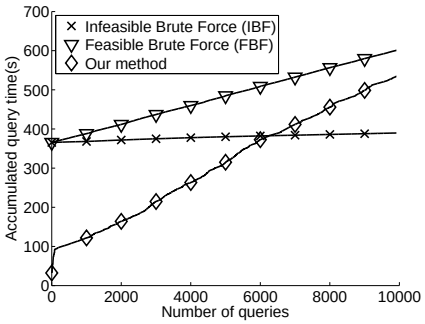


Figure 8: Cumulative cost in a workload

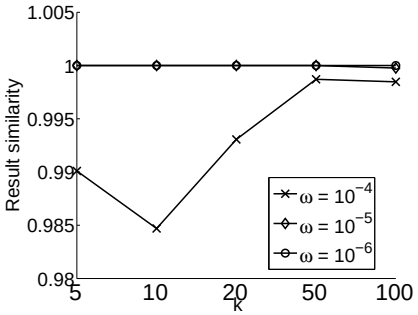


Figure 9: Effect of rounding

author	reverse top-5 size	# coauthors
Philip S. Yu	2020	231
Jiawei Han	2007	253
Christos Faloutsos	1932	221
Zheng Chen	162	137
Qiang Yang	161	166
Daphne Koller	157	98
C. Lee Giles	155	132
Gerhard Weikum	149	130
Michael I. Jordan	147	125
Bernhard Schölkopf	140	134

Table 3: Longest reverse top-5 lists of DBLP authors

types of web hosts give their top-5 PageRank contributions to each query node. Our experimental results show that if a query web host is classified as spam, on average 96.1% web hosts in its reverse top-5 set are also spam nodes; on the other hand, if the query is a normal web host, on average 97.4% web hosts in its reverse top-5 result are normal. Therefore, reverse top-k results using RWR are a very strong indicator toward detection of spam web hosts. In a real scenario, we can apply a reverse top-k RWR search on any suspicious web host, and make a judgement according to the spam ratio of the labeled answer set.

Popularity of authors in a coauthorship network. The size of a reverse top-k query can also be an indicator of the popularity of the query node in the graph. We extracted from DBLP[6] the publications in top venues in the fields of databases, data mining, machine learning, artificial intelligence, computer vision, and information retrieval. We generated a coauthorship network, with 44528 nodes and 121352 edges where each node corresponds to an author and an edge indicates coauthorship. To reflect the different weights in coauthorships, we changed the RWR transition matrix as follows:

$$\boldsymbol{a}_{i,j} = \begin{cases} \frac{w_{i,j}}{w_j} & \text{if edge } j \rightarrow i \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

where w_j is the number of publications of author j and $w_{i,j}$ is the number of papers that i and j coauthored. We carried out reverse top-5 search from all the nodes in the graph, and obtained a descending ranked list of authors w.r.t. the size of their answer set. The 10 authors with the longest reverse top-5 lists are shown in Table 3. The table indicates that there are three "popular" authors with very long reverse top-5 lists, which stand out.[7] More importantly, the reverse top-k lists of these three authors are much longer than their coauthor lists (third column of Table 3), which indicates that there are many non-coauthors having them in their reverse top-k sets. Therefore, the size of a reverse top-k query can be a stronger indicator for popularity, compared to the node's degree.

[6]dblp.uni-trier.de/xml/

6. RELATED WORK

6.1 Random Walk with Restart

Random work with restart has been a widely used node-to-node proximity in graph data, especially after its successful application by the search engine Google [21] to derive the importance (i.e., PageRank) of web pages.

Early works focused on how to efficiently solve the linear system (1). Although non-iterative methods such as Gaussian elimination can be applied, their high complexity of $O(n^3)$ makes them unaffordable in real scenarios. Iterative approaches such as the Power Method (PM) [21] and Jacobi algorithm have a lower complexity of $O(Dm)$, where $D(\ll n < m)$ is the number of iterations. Later on, faster (but less accurate) methods such as Hub-vector decomposition [15] have been proposed. As this method restricts the restarting only to a specific node set, it does not compute exactly the proximity vectors of all nodes in the graph.

To further accelerate the computation of RWR, approximate approaches have been introduced. [22] leverages the block structure of the graph and only calculates the RWR similarity within the partition containing the query node. Later, Monte Carlo (MC) methods are introduced to simulate the random walk process, such as [9, 3, 18]. The simulation can be stored as a fingerprint for fast online RWR estimation. Recently, a scheduled approximation strategy is proposed by [30] to compute RWR proximity. From another viewpoint of RWR, Bookmark Coloring Algorithm (BCA) [7] has been proposed to derive a sparse lower bound approximation of the real proximity vector (see Section 2 for details). Our offline index is based on approximations derived by partial execution of BCA and not on other approaches, such as PM or MC simulation, because the latter do not guarantee that their approximations are lower bounds of the exact proximities and therefore do not fit into our framework of using lower and upper proximity bounds to accelerate search.

[7]By "popular" here we mean authors who are likely to be approachable by many other authors and intuitively have higher chance to collaborate with them in the future. Indeed, there are many other authors who are very popular (e.g., in terms of visibility) and they do not show up in Table 3, but these authors are likely to work in smaller groups and do not have so much open collaboration, compared to those having larger reverse top-k sets.

6.2 Top-k RWR Proximity Search

Bahmani et al. [4] observed that the majority of entries in a proximity vector are extremely small. Thus, in many cases, it is unnecessary to compute the exact RWR proximity from the query node to all remaining nodes, especially to those with extremely low proximities. Based on this observation, several top-k proximity search algorithms are introduced. Based on BCA [7], [11] proposed the Basic Push Algorithm (BPA). At each iteration, BPA maintains a set of top-k candidates and estimates an upper bound for the $(k+1)$-th largest proximity. BPA stops as soon as the upper bound is not greater than the current k-th largest proximity. Recently, another method, K-dash [10] was proposed. In an indexing stage, K-dash applies LU decomposition on the proximity matrix $\boldsymbol{P}$ and stores the sparse matrices $\boldsymbol{L}^{-1}$ and $\boldsymbol{U}^{-1}$. In the query stage, it builds a BFS tree rooted at the query node and estimates an upper bound for each visited node. Such estimation can help determine whether K-dash should continue or terminate.

When the exact order of the top-k list is not important and a few misplaced elements are acceptable, Monte Carlo methods can be used to simulate RWR from the query node u. [3] designs two such algorithms; MC End Point and MC Complete Path. The former evaluates RWR proximity $\boldsymbol{p}_u(v)$ as the fraction of t random walks which end at node v, while the latter evaluates $\boldsymbol{p}_u(v)$ as the number of visits to node v multiplied by $(1-c)/t$.

6.3 Reverse k-NN and Reverse Top-k Search

Reverse k nearest neighbors (RkNN) search aims at finding *all* objects in a set T that have a given query object from T in their k-NN sets. In the Euclidean space, RkNN queries were introduced in [17]; an efficient geometric solution was proposed in [24]. RkNN search has also been studied for objects lying on large graphs, albeit using shortest path as the proximity measure, which makes the problem much easier [28]. The reverse top-k query is defined by Vlachou et al. in [26] as follows. Given a set of multi-dimensional data points and a query point q, the goal is to find all linear preference functions that define a total ranking in the data space such that q lies in the top-k result of the functions. Solutions for RkNN and reverse top-k queries cannot be applied to solve our problem, due to the special nature of graph data and/or the use of RWR proximity.

7. CONCLUSIONS

In this paper, we have studied for the first time the problem of reverse top-k proximity search in large graphs, based on the random walk with restart (RWR) measure. We showed that the naive evaluation of this problem is too expensive and proposed an index which keeps track of lower bounds for the top proximity values from each node. Our online query evaluation technique first computes the exact RWR proximities from the query node q to all graph nodes and then compares them with the top-k lower bounds derived from the index. For nodes that cannot be pruned, we compute upper bounds for their k-th proximities and use them to test whether they are in the reverse top-k result. For any remaining candidates, their k-th proximity lower and upper bounds are progressively refined until they become results or they are pruned. Our experiments confirm the efficiency of our approach; in addition we demonstrate the use of reverse top-k queries in identifying spam web hosts or popular authors in co-authorship networks. As future work, we plan to generalize the problem of reverse top-k search to other proximity measures such as SimRank [14]. Since the current framework does not consider the dynamics of the graph, we would also like to extend our method to do reverse top-k search on evolving graphs. The key challenge is how to maintain the index incrementally.

8. REFERENCES

[1] R. Andersen, C. Borgs, J. T. Chayes, J. E. Hopcroft, V. S. Mirrokni, and S.-H. Teng. Local computation of pagerank contributions. In *WAW*, 2007.

[2] R. Andersen, F. R. K. Chung, and K. J. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, 2006.

[3] K. Avrachenkov, N. Litvak, D. Nemirovsky, E. Smirnova, and M. Sokol. Quick detection of top-k personalized pagerank lists. In *WAW*, 2011.

[4] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *PVLDB*, 4(3):173–184, 2010.

[5] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, 2004.

[6] A. A. Benczúr, K. Csalogány, T. Sarlós, and M. Uher. Spamrank – fully automatic link spam detection. In *AIRWeb*, 2005.

[7] P. Berkhin. Bookmark-coloring approach to personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.

[8] Y.-Y. Chen, Q. Gan, and T. Suel. Local methods for estimating pagerank values. In *CIKM*, 2004.

[9] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[10] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa. Fast and exact top-k search for random walk with restart. *PVLDB*, 5(5):442–453, 2012.

[11] M. S. Gupta, A. Pathak, and S. Chakrabarti. Fast algorithms for topk personalized pagerank queries. In *WWW*, 2008.

[12] T. H. Haveliwala. Topic-sensitive pagerank. In *WWW*, 2002.

[13] J. He, M. Li, H. Zhang, H. Tong, and C. Zhang. Manifold-ranking based image retrieval. In *ACM Multimedia*, 2004.

[14] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *KDD*, 2002.

[15] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, 2003.

[16] I. Konstas, V. Stathopoulos, and J. M. Jose. On social networks and collaborative recommendation. In *SIGIR*, 2009.

[17] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD Conference*, 2000.

[18] N. Li, Z. Guan, L. Ren, J. Wu, J. Han, and X. Yan. giceberg: Towards iceberg analysis in large graphs. In *ICDE*, 2013.

[19] D. Liben-Nowell and J. M. Kleinberg. The link prediction problem for social networks. In *CIKM*, 2003.

[20] A. Y. Ng, A. X. Zheng, and M. I. Jordan. Link analysis, eigenvectors and stability. In *IJCAI*, 2001.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[22] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos. Neighborhood formation and anomaly detection in bipartite graphs. In *ICDM*, 2005.

[23] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB*, 4(11), 2011.

[24] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, 2004.

[25] H. Tong, C. Faloutsos, and Y. Koren. Fast direction-aware proximity for graph mining. In *KDD*, 2007.

[26] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, 2010.

[27] J. H. Wilkinson. *The algebraic eigenvalue problem*, volume 155. Oxford Univ Press, 1965.

[28] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. In *ICDE*, 2005.

[29] A. W. Yu, N. Mamoulis, and H. Su. Reverse top-k search using random walk with restart. Technical Report TR-2013-08, CS Department, HKU, September 2013.

[30] F. Zhu, Y. Fang, K. C.-C. Chang, and J. Ying. Incremental and accuracy-aware personalized pagerank through scheduled approximation. *PVLDB*, 6(6), 2013.