Random Walk with Restart on Large Graphs Using Block Elimination

JINHONG JUNG, Seoul National University
KIJUNG SHIN, Carnegie Mellon University
LEE SAEL, State University of New York (SUNY) Korea
U KANG, Seoul National University

Given a large graph, how can we calculate the relevance between nodes fast and accurately? Random walk with restart (RWR) provides a good measure for this purpose and has been applied to diverse data mining applications including ranking, community detection, link prediction, and anomaly detection. Since calculating RWR from scratch takes a long time, various preprocessing methods, most of which are related to inverting adjacency matrices, have been proposed to speed up the calculation. However, these methods do not scale to large graphs because they usually produce large dense matrices that do not fit into memory. In addition, the existing methods are inappropriate when graphs dynamically change because the expensive preprocessing task needs to be computed repeatedly.

In this article, we propose BEAR, a fast, scalable, and accurate method for computing RWR on large graphs. BEAR has two versions: a preprocessing method BEARS for static graphs and an incremental update method BEARD for dynamic graphs. BEARS consists of the preprocessing step and the query step. In the preprocessing step, BEARS reorders the adjacency matrix of a given graph so that it contains a large and easy-to-invert submatrix, and precomputes several matrices including the Schur complement of the submatrix. In the query step, BEARS quickly computes the RWR scores for a given query node using a block elimination approach with the matrices computed in the preprocessing step. For dynamic graphs, BEARD efficiently updates the changed parts in the preprocessed matrices of BEARS based on the observation that only small parts of the preprocessed matrices change when few edges are inserted or deleted. Through extensive experiments, we show that BEARS significantly outperforms other state-of-the-art methods in terms of preprocessing and query speed, space efficiency, and accuracy. We also show that BEARD quickly updates the preprocessed matrices and immediately computes queries when the graph changes.

Categories and Subject Descriptors: H.2.8 [**Database management**]: Database Applications—*Data mining*

General Terms: Design, Experimentation, Algorithms

Additional Key Words and Phrases: Proximity, ranking in graph, random walk with restart, relevance score

12

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) and funded by the Ministry of Science, ICT and Future Planning (NRF-2013R1A1A3005259), and IT R&D program of MSIP/IITP (10044970, "Development of Core Technology for Human-Like Self-Taught Learning Based on Symbolic Approach"). This work was also funded by an Institute for Information Communications Technology Promotion (IITP) grant funded by the Korean government (MSIP) (R0190-15-2012, "High Performance Big Data Analytics Platform Performance Acceleration Technologies Development"). The ICT at Seoul National University provided research facilities for this study.
Authors' addresses: J. Jung, Department of Computer Science and Engineering, Seoul National University; email: jinhongjung@snu.ac.kr; K. Shin, Computer Science Department, Carnegie Mellon University; email: kijungs@cs.cmu.edu; L. Sael, Department of Computer Science, The State University of New York (SUNY) Korea; email: sael@sunykorea.ac.kr; U Kang (corresponding author), Department of Computer Science and Engineering, Seoul National University; email: ukang@snu.ac.kr.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0362-5915/2016/05-ART12 $15.00
DOI: http://dx.doi.org/10.1145/2901736

ACM Reference Format:
Jinhong Jung, Kijung Shin, Lee Sael, and U Kang. 2016. Random walk with restart on large graphs using block elimination. ACM Trans. Database Syst. 41, 2, Article 12 (May 2016), 43 pages.
DOI: http://dx.doi.org/10.1145/2901736

1. INTRODUCTION

Measuring the relevance (proximity) between nodes in a graph becomes the base for various data mining tasks [Andersen et al. 2006; Backstrom and Leskovec 2011; Gleich and Seshadhri 2012; He et al. 2004; Liben-Nowell and Kleinberg 2007; Sun et al. 2005; Tong et al. 2007; Tong and Faloutsos 2006; Whang et al. 2013; Zhang et al. 2012; Zhu et al. 2013] and has received much interest from the database research community [Antonellis et al. 2008; Chakrabarti et al. 2011; Fujiwara et al. 2012a; Wu et al. 2014]. Among many methods [Adamic and Adar 2003; Jeh and Widom 2002; Lin et al. 2009; Pan et al. 2004] to compute the relevance, random walk with restart (RWR) [Pan et al. 2004] has been popular due to its ability to account for the global network structure [He et al. 2004] and the multifaceted relationship between nodes [Tong and Faloutsos 2006]. RWR has been used in many data mining applications, including ranking [Tong et al. 2008], community detection [Andersen et al. 2006; Gleich and Seshadhri 2012; Whang et al. 2013; Zhu et al. 2013], link prediction [Backstrom and Leskovec 2011], and anomaly detection [Sun et al. 2005].

However, existing methods for computing RWR are unsatisfactory in terms of speed, accuracy, functionality, and scalability. The iterative method, which naturally follows from the definition of RWR, is not fast: it requires repeated matrix-vector multiplications whose computational cost is not acceptable in real-world applications where RWR scores for different query nodes need to be computed. Several approximate methods [Andersen et al. 2006; Gleich and Polito 2006; Sun et al. 2005; Tong et al. 2008] have also been proposed; however, their accuracies or speedups are unsatisfactory. Although top-k methods [Fujiwara et al. 2012a; Wu et al. 2014] improve efficiency by focusing on finding the k most relevant nodes and ignoring irrelevant ones, finding top-k is insufficient for many data mining applications [Andersen et al. 2006; Backstrom and Leskovec 2011; Gleich and Polito 2006; He et al. 2004; Sun et al. 2005; Tong et al. 2007; Whang et al. 2013; Zhu et al. 2013] that require the relevance scores of all nodes or the least relevant nodes. Existing preprocessing methods [Fujiwara et al. 2012a, 2012b], which achieve better performance by preprocessing the given graph, are not scalable due to their high memory requirements. Moreover, those preprocessing methods are inappropriate when the given graph changes over time since the expensive preprocessing task should be repeated.

In this article, we propose BEAR, a fast, scalable, and accurate method for computing RWR on large graphs. BEAR comprises two methods: a preprocessing method BEARS for static graphs and an incremental update method BEARD for dynamic graphs. In the preprocessing step, BEARS reorders the adjacency matrix of a given graph so that it contains a large and easy-to-invert submatrix, and precomputes several matrices, including the Schur complement [Boyd and Vandenberghe 2009] of the submatrix. In the query step, BEARS computes the RWR scores for a given query node, quickly using a block elimination approach with the matrices computed in the preprocessing step. BEARS has two versions: an exact method BEARS-EXACT and an approximate method BEARS-APPROX. The former provides accuracy assurance; the latter gives faster query speed and requires less space by allowing small accuracy loss. For dynamic graphs, BEARD incrementally updates the preprocessed matrices from BEARS based on the observation that only small parts of the preprocessed matrices change when few edges are inserted or deleted.

Through extensive experiments with various real-world datasets, we demonstrate the superiority of BEARS over other state-of-the-art methods. In addition, we present that BEARD quickly updates the preprocessed matrices when the graph changes and immediately computes RWR scores for given queries. We also discuss how our method can be applied to other random walk–based measures, such as personalized PageRank [Page et al. 1999]; effective importance (EI) [Bogdanov and Singh 2013]; RWR with normalized graph Laplacian [Tong et al. 2008]; and other graph similarities, such as FaBP [Koutra et al. 2011]. The main characteristics of our method are the following:

—*Fast*: BEARS-EXACT is faster up to 8× in the query phase and up to 12× in the preprocessing phase than other exact methods (Figures 5(a) and 6), and BEARS-APPROX achieves a better time/accuracy trade-off in the query phase than other approximate methods (Figure 11). Updating the preprocessed matrices using BEARD is up to 29× faster than preprocessing the changed graphs from the beginning (Figure 12(a)), and BEARD quickly computes RWR scores for queries using the updated matrices (Figure 12(b)).
—*Space efficient*: Compared to their respective competitors, BEARS-EXACT requires up to 22× less memory space (Figure 5(b)), and BEARS-APPROX provides a better space/accuracy trade-off (Figure 11).
—*Accurate*: BEARS-EXACT guarantees exactness (Theorem 3.4); BEARS-APPROX enjoys a better trade-off between accuracy, time, and space than other approximate methods (Figure 11). BEARD updates the preprocessed matrices and computes exact RWR scores for queries (Theorem 4.5).
—*Versatile*: BEAR can be applied to diverse RWR variants, including personalized PageRank (PPR), EI, RWR with normalized graph Laplacian, and fast belief propagation (FaBP) (Section 3.4).

The codes of our methods and datasets are available at http://datalab.snu.ac.kr/bear. The rest of the article is organized as follows. Section 2 presents preliminaries on RWR. Our proposed preprocessing method BEARS for static graphs is described in Section 3, and our incremental update method BEARD for dynamic graphs is proposed in Section 4. We demonstrate the experimental results in Section 5. After providing a review on related work in Section 6, we offer our conclusion in Section 7.

2. PRELIMINARIES

In this section, we describe the preliminaries on RWR and its algorithms. Table I lists the symbols used in this article. We denote matrices by boldface capital letters (e.g., $\mathbf{A}$) and vectors by boldface lowercase letters (e.g., $\mathbf{q}$).

2.1. Random Walk with Restart

RWR [Tong et al. 2008] measures each node's relevance with respect to a given seed node s in a given graph. It assumes a random surfer who occasionally gets bored with following the edges in the graph and restarts at node s. The surfer starts at node s, and at each current node either restarts at node s (with probability c) or moves to a neighboring node along an edge (with probability $1 - c$). The probability that each edge is chosen is proportional to its weight in the adjacency matrix. $\tilde{\mathbf{A}}$ denotes the row-normalized adjacency matrix, whose (u, v)th entry is the probability that a surfer at node u chooses the edge to node v among all edges from node u. The stationary probability of being at each node corresponds to its RWR score with respect to node s and is denoted by $\mathbf{r}$, whose uth entry corresponds to node u's RWR score. The vector $\mathbf{r}$

Table I. Table of Symbols

Symbol	Definition
G	input graph
n	number of nodes in G
m	number of edges in G
n_1	number of spokes in G
n_2	number of hubs in G
n_{1i}	number of nodes in the ith diagonal block of $\mathbf{H_{11}}$
b	number of diagonal blocks in $\mathbf{H_{11}}$
s	seed node (=query node)
c	restart probability
ξ	drop tolerance
$\mathbf{D}$	$(n \times n)$ diagonal matrix of degrees, $\mathbf{D}_{ii} = \sum_j \mathbf{A}_{ij}$
$\mathbf{A}$	$(n \times n)$ unnormalized adjacency matrix of G
$\tilde{\mathbf{A}}$	$(n \times n)$ row-normalized adjacency matrix of G
$\mathbf{H}$	$(n \times n)$ $\mathbf{H} = \mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}$
$\mathbf{H_{ij}}$	$(n_i \times n_j)$ (i, j)th partition of $\mathbf{H}$
$\mathbf{S}$	$(n_2 \times n_2)$ Schur complement of $\mathbf{H_{11}}$
$\mathbf{L_1}, \mathbf{U_1}$	$(n_1 \times n_1)$ LU-decomposed matrices of $\mathbf{H_{11}}$
$\mathbf{L_2}, \mathbf{U_2}$	$(n_2 \times n_2)$ LU-decomposed matrices of $\mathbf{S}$
$\mathbf{q}$	$(n \times 1)$ starting vector
$\mathbf{q_i}$	$(n_i \times 1)$ ith partition of $\mathbf{q}$
$\mathbf{r}$	$(n \times 1)$ relevance vector
$\mathbf{r_i}$	$(n_i \times 1)$ ith partition of $\mathbf{r}$
T	number of SlashBurn iterations
k	number of hubs removed at a time in SlashBurn
t	rank in B_LIN and NB_LIN
ϵ	threshold to stop iteration in iterative methods
ϵ_b	threshold to expand nodes in RPPR and BRPPR
(u, v, w)	edge modification, edge (u, v) with weight w
$\hat{\mathbf{A}}$	updated matrix of $\mathbf{A}$ after one edge modification
$\Delta\mathbf{A}$	difference matrix of $\mathbf{A}$, $\hat{\mathbf{A}} - \mathbf{A}$, after one edge modification
$\mathbf{H}_{11}^{i}$	$(n_{1i} \times n_{1i})$ block matrix of an index i in $\mathbf{H}_{11}$
i_u	index of a block to which node u belongs
$N(u)$	set of out-neighbors of node u
$\Gamma(\mathbf{A}, i)$	block contribution of one block i to the Schur complement of $\mathbf{A}_{11}$
$\backslash_F, \backslash_B$	backslash operators for forward and backward substitution algorithms

satisfies the following equations:

$$\begin{aligned} \tilde{\mathbf{A}} &= \mathbf{D}^{-1}\mathbf{A}, \\ \mathbf{r} &= (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}\mathbf{r} + c\mathbf{q}, \end{aligned} \tag{1}$$

where $\mathbf{D}$ is the diagonal matrix of degrees (i.e., $\mathbf{D}_{ii} = \sum_j \mathbf{A}_{ij}$) and $\mathbf{q}$ is the starting vector in which the index of the seed node s is set to 1 and others to 0. It can be obtained by solving the following linear equation:

$$\begin{aligned} &(\mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}})\mathbf{r} = c\mathbf{q} \\ &\Leftrightarrow \mathbf{H}\mathbf{r} = c\mathbf{q}. \end{aligned} \tag{2}$$

Personalized Page-Rank. PPR [Page et al. 1999] is an extension of RWR. PPR calculates the relevance of nodes according to the preference of each user and is widely

used for personalized search. A random surfer in PPR either jumps to a random node according to the probability distribution (user preference distribution) given by $\mathbf{q}$ (with probability c) or moves to an out-neighbor (with probability $1 - c$). PPR can be viewed as a generalized version of RWR with multiple seed nodes. Equations (1) and (2) can be directly applied to PPR with the modified $\mathbf{q}$.

2.2. Algorithms for RWR

We review two basic methods for RWR computation and four recent methods for addressing the limitations of the basic methods. We also point out a need for improvement, which we will address in the following section. Since most applications require RWR scores for different seed nodes, whether at once or on demand, we separate the preprocessing phase, which occurs once, from the query phase, which occurs for each seed node.

Iterative method. The iterative method repeats updating $\mathbf{r}$ until convergence ($|\mathbf{r}^{(i)} - \mathbf{r}^{(i-1)}| < \epsilon$) by the following update rule:

$$\mathbf{r}^{(i)} \leftarrow (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}\mathbf{r}^{(i-1)} + c\mathbf{q}, \tag{3}$$

where the superscript i denotes the iteration number. If $0 < c < 1$, $\mathbf{r}^{(i)}$ is guaranteed to converge to a unique solution [Langville and Meyer 2011]. This method does not require preprocessing (one-time cost) but has expensive query cost that incurs a repeated matrix-vector multiplication. Thus, it is inefficient when RWR scores for many query nodes are required.

RPPR/BRPPR. Gleich and Polito [2006] propose restricted personalized Page-Rank (RPPR), which speeds up the iterative method by accessing only a part of a graph. This algorithm uses Equation (3) only for a subgraph and the nodes contained in it. The subgraph is initialized to a given seed node and grows as iteration proceeds. A node contained in the subgraph is on the boundary if its outgoing edges (in the original graph) are not contained in the subgraph, and the outgoing edges and outgoing neighbors of the node are added to the subgraph if the RWR score of the node (in the current iteration) is greater than a threshold ϵ_b. The algorithm repeats iterations until RWR scores converge. The RWR scores of nodes outside the subgraph are set to zero. Boundary-restricted personalized Page-Rank (BRPPR) [Gleich and Polito 2006] is a variant of the RPPR. It expands nodes on the boundary in decreasing order of their RWR scores (in the current iteration) until the sum of the RWR scores of nodes on the boundary becomes less than a threshold ϵ_b. Although these methods reduce the query cost of the iterative method significantly, they do not guarantee exactness.

Push. Andersen et al. [2006] propose an ϵ-approximate PPR computation method (Push) on undirected graphs, which iteratively computes PPR scores. The method first starts with a PPR score vector, $\mathbf{r} = 0$, and a residual score vector, $\mathbf{k} = \mathbf{q}$. Then, the method repeats a series of push operations that move probability from $\mathbf{k}$ to $\mathbf{r}$ until for each node u, $\mathbf{k}(u) < \epsilon\mathbf{d}(u)$ is satisfied, where $\mathbf{d}(u)$ is a degree of node u. As with BPPR or BRPPR, Push computes PPR scores quickly, but the exactness of the method is not guaranteed.

Inversion. An algebraic method directly calculates $\mathbf{r}$ from Equation (2) as follows:

$$\mathbf{r} = c(\mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}})^{-1}\mathbf{q} = c\mathbf{H}^{-1}\mathbf{q}. \tag{4}$$

The matrix $\mathbf{H}$ is known to be invertible when $0 < c < 1$ [Langville and Meyer 2011]. Once $\mathbf{H}^{-1}$ is computed in the preprocessing phase, $\mathbf{r}$ can be obtained efficiently in the query phase. However, this is again impractical for large graphs because calculating $\mathbf{H}^{-1}$ is computationally expensive and $\mathbf{H}^{-1}$ is usually too dense to fit into memory as shown later in Section 5 (see Figure 4(a)).

QR decomposition. To avoid the problem regarding $\mathbf{H}^{-1}$, Fujiwara et al. [2012b] decompose $\mathbf{H}$ using QR decomposition and then use $\mathbf{Q}^{\mathbf{T}}(=\mathbf{Q}^{-1})$ and $\mathbf{R}^{-1}$ instead of $\mathbf{H}^{-1}$ as follows:

$$\mathbf{r} = c\mathbf{H}^{-1}\mathbf{q} = c\mathbf{R}^{-1}(\mathbf{Q}^{\mathbf{T}}\mathbf{q}),$$

where $\mathbf{H} = \mathbf{QR}$. They also propose a reordering rule for $\mathbf{H}$ that makes $\mathbf{Q}^{\mathbf{T}}$ and $\mathbf{R}^{-1}$ sparser. However, on the most datasets used in our experiments, QR decomposition results in dense matrices as shown in Section 5 (see Figure 4(b) and (c)); thus, its scalability is limited. This fact agrees with the claim made by Boyd and Vandenberghe [2009] that it is difficult to exploit sparsity in QR decomposition.

LU decomposition. To replace $\mathbf{H}^{-1}$, Fujiwara et al. [2012a] also exploit LU decomposition using the following rule:

$$\mathbf{r} = c\mathbf{H}^{-1}\mathbf{q} = c\mathbf{U}^{-1}(\mathbf{L}^{-1}\mathbf{q}),$$

where $\mathbf{H} = \mathbf{LU}$. Prior to the decomposition, $\mathbf{H}$ is reordered based on nodes' degrees and community structure. This makes matrices $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$ sparser as shown in Section 5 (see Figure 4(d) and (e)). We incorporate their idea into our method to replace inverse terms, which will be explained in detail in Section 3.

B_LIN/NB_LIN. Tong et al. [2008] partition a given graph and divide $\tilde{\mathbf{A}}^{\mathbf{T}}$ into $\mathbf{A_1}$ (inner-partition edges) and $\mathbf{A_2}$ (cross-partition edges). Then, they use a heuristic decomposition method with given rank t to approximate $\mathbf{A_2}$ with low-rank matrix $\mathbf{U\Sigma V}$, where $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}$ are $n \times t$, $t \times t$, and $t \times n$ matrices, respectively. In the query phase, they apply the Sherman-Morrison lemma [Piegorsch and Casella 1990] to efficiently calculate $\mathbf{r}$ as follows:

$$\begin{aligned}\mathbf{r} &= c(\mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}})^{-1}\mathbf{q}\\ &\approx c(\mathbf{I} - (1-c)\mathbf{A_1} - (1-c)\mathbf{U\Sigma V})^{-1}\mathbf{q}\\ &= c\big(\mathbf{A_1^{-1}}\mathbf{q} + (1-c)\mathbf{A_1^{-1}}\mathbf{U}\tilde{\mathbf{\Lambda}}\mathbf{V}\mathbf{A_1^{-1}}\mathbf{q}\big),\end{aligned}$$

where $\tilde{\mathbf{\Lambda}} = (\mathbf{\Sigma}^{-1} - c\mathbf{V}\mathbf{A_1^{-1}}\mathbf{U})^{-1}$. To sparsify the precomputed matrices, near-zero entries whose absolute value is smaller than ξ are dropped. This method is called *B_LIN*, and its variant *NB_LIN* directly approximates $\tilde{\mathbf{A}}^{\mathbf{T}}$ without partitioning it. Both methods do not guarantee exactness.

As summarized later in Figure 4, previous preprocessing methods require too much space for preprocessed data or do not guarantee accuracy. Our proposed BEARS, explained in the following section, achieves both space efficiency and accuracy as seen later in Figure 4(i) through (k).

3. PROPOSED METHOD FOR STATIC GRAPHS

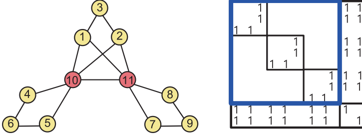
In this section, we describe BEARS, our proposed method for fast, scalable, and accurate RWR computation. BEARS has two versions: BEARS-EXACT for exact RWR and BEARS-APPROX for approximate RWR. BEARS-EXACT guarantees accuracy, whereas BEARS-APPROX improves speed and space efficiency by sacrificing little accuracy. A pictorial description of BEARS is provided in Figure 1. BEARS exploits the following ideas:

—The adjacency matrix of real-world graphs can be reordered so that it has a large but easy-to-invert submatrix, such as a block-diagonal matrix, as shown in the upper-left part of Figure 2(b).

—A linear equation like Equation (2) is easily solved by block elimination using the Schur complement if the matrix contains a large and easy-to-invert submatrix such as a block-diagonal one.

Preprocessing phase:

(1) Node Reordering



(2) Partitioning

(3) Schur Complement

(4) LU Decomposition

Query phase:

(5) Block Elimination

Fig. 1. Pictorial description of BEARS. The output matrices of the preprocessing phase are bordered in red. (1) Reorder nodes so that the adjacency matrix has a large block-diagonal submatrix (the blue-bordered one). (2) Partition $\mathbf{H}$ into four blocks so that $\mathbf{H_{11}}$ corresponds to the submatrix. (3) Compute the Schur complement $\mathbf{S}$ of $\mathbf{H_{11}}$. (4) Since $\mathbf{S^{-1}}$ and the diagonal blocks of $\mathbf{H_{11}^{-1}}$ are likely to be dense, store the inverse of the LU decomposed matrices of $\mathbf{S}$ and $\mathbf{H_{11}}$ instead to save space. Notice that these can be computed efficiently in terms of time and space because $\mathbf{S}$ and the diagonal blocks of $\mathbf{H_{11}}$ are relatively small compared with $\mathbf{H}$. (5) Compute the RWR score vector $\mathbf{r}$ for a query vector $\mathbf{q}$ (the concatenation of $\mathbf{q_1}$ and $\mathbf{q_2}$) fast by utilizing the precomputed matrices.

—Compared to directly inverting an adjacency matrix, inverting its LU-decomposed matrices is more efficient in terms of time and space.

The reason BEARS exploits block elimination is that if a submatrix of the given matrix is inverted easily, then the inverse of the given matrix is efficiently computed by block elimination. As mentioned in the first idea, the reordered adjacency matrix of most real-world graphs contains a large but easy-to-invert submatrix, and thus the linear system based on the adjacency matrix such as Equation (4) is efficiently solved by block elimination.

Algorithms 1 and 2 represent the procedure of BEARS. Since RWR scores are requested for different seed nodes in real-world applications, we separate the preprocessing phase (Algorithm 1), which is run once, from the query phase (Algorithm 2), which is run for each seed node. The exact method BEARS-EXACT and the approximate method BEARS-APPROX differ only at line 9 of Algorithm 1; the detail is provided in Section 3.1.4. To exploit their sparsity, all matrices considered are stored in a sparse matrix format, such as the compressed sparse column format [Press 2007], which stores only nonzero entries.

ALGORITHM 1: Preprocessing Phase in BEARS

Input: graph: G, restart probability: c, drop tolerance: ξ
Output: precomputed matrices: $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, $\mathbf{U_2^{-1}}$, $\mathbf{H_{12}}$, and $\mathbf{H_{21}}$

1: compute $\tilde{\mathbf{A}}$, and $\mathbf{H} = \mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}$
2: find hubs using SlashBurn [Kang and Faloutsos 2011], and divide spokes into disconnected components by removing the hubs
3: reorder nodes and $\mathbf{H}$ so that the disconnected components form a block-diagonal submatrix $\mathbf{H_{11}}$ where each block is ordered in the ascending order of degrees within the component
4: partition $\mathbf{H}$ into $\mathbf{H_{11}}, \mathbf{H_{12}}, \mathbf{H_{21}}, \mathbf{H_{22}}$
5: decompose $\mathbf{H_{11}}$ into $\mathbf{L_1}$ and $\mathbf{U_1}$ using LU decomposition and compute $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$
6: compute the Schur complement of $\mathbf{H_{11}}$, $\mathbf{S} = \mathbf{H_{22}} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}(\mathbf{H_{12}})))$
7: reorder the hubs in the ascending order of their degrees in $\mathbf{S}$ and reorder $\mathbf{S}$, $\mathbf{H_{21}}$, and $\mathbf{H_{12}}$ according to it
8: decompose $\mathbf{S}$ into $\mathbf{L_2}$ and $\mathbf{U_2}$ using LU decomposition and compute $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$
9: (BEARS-APPROX only) drop entries whose absolute value is smaller than ξ in $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, $\mathbf{U_2^{-1}}$, $\mathbf{H_{12}}$, and $\mathbf{H_{21}}$
10: **return** $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, $\mathbf{U_2^{-1}}$, $\mathbf{H_{12}}$, and $\mathbf{H_{21}}$

3.1. Preprocessing Phase

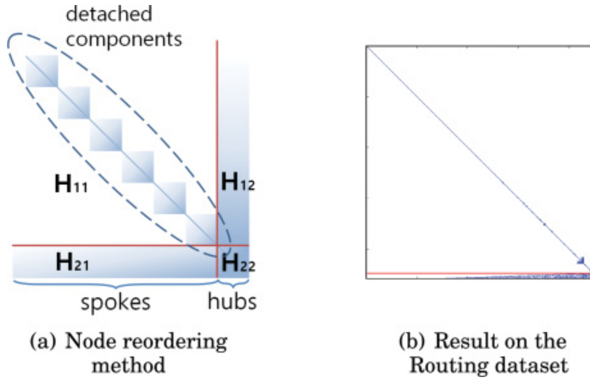
The overall preprocessing phase of BEARS is shown in Algorithm 1, and the details are explained in the following sections.

3.1.1. Node Reordering (Lines 2 Through 4). In this part, we reorder $\mathbf{H}(= \mathbf{I}-(1-c)\tilde{\mathbf{A}}^{\mathbf{T}})$ and partition it. Our objective is to reorder $\mathbf{H}$ so that it has a large but easy-to-invert submatrix such as a block-diagonal one. Any node reordering method (e.g., spectral clustering [Ng et al. 2002], cross association [Chakrabarti et al. 2004a], shingle [Chierichetti et al. 2009]) can be used for this purpose; in this article, we use a method that improves on SlashBurn [Kang and Faloutsos 2011] since it is the state-of-the-art method in concentrating the nonzeros of adjacency matrices of graphs (more details are provided in Appendix A.1). We first run SlashBurn on a given graph to decompose the graph into hubs (high-degree nodes) and spokes (low-degree nodes that get disconnected from the giant connected component (GCC) if the hubs are removed). Within each connected component containing spokes, we reorder nodes in the ascending order of degrees within the component. As a result, we get an adjacency matrix whose upper-left area (e.g., $\mathbf{H_{11}}$ in Figure 2(a)) is a large and sparse block-diagonal matrix that is easily inverted, whereas the lower-right area (e.g., $\mathbf{H_{22}}$ in Figure 2(a)) is a small but dense matrix. Let n_1 denote the number of spokes and n_2 denote the number of hubs. After the reordering, we partition the matrix $\mathbf{H}$ into four pieces: $\mathbf{H_{11}}$ ($n_1 \times n_1$ matrix), $\mathbf{H_{12}}$ ($n_1 \times n_2$ matrix), $\mathbf{H_{21}}$ ($n_2 \times n_1$ matrix), and $\mathbf{H_{22}}$ ($n_2 \times n_2$ matrix), which correspond to the adjacency matrix representation of edges between spokes, from spokes to hubs, from hubs to spokes, and between hubs, respectively.

3.1.2. Schur Complement (Lines 5 Through 6). In this part, we compute the Schur complement of $\mathbf{H_{11}}$, whose inverse is required in the query phase.

Definition 3.1 (*Schur Complement [Boyd and Vandenberghe 2009]*). Suppose that a square matrix $\mathbf{A}$ is partitioned into $\mathbf{A_{11}}, \mathbf{A_{12}}, \mathbf{A_{21}}$, and $\mathbf{A_{22}}$, which are $p\times p$, $p\times q$, $q\times p$, and $q\times q$ matrices, respectively, and $\mathbf{A_{11}}$ is invertible. The Schur complement $\mathbf{S}$ of the block $\mathbf{A_{11}}$ of the matrix $\mathbf{A}$ is defined by

$$\mathbf{S} = \mathbf{A_{22}} - \mathbf{A_{21}}\mathbf{A_{11}^{-1}}\mathbf{A_{12}}.$$



(a) Node reordering method

(b) Result on the Routing dataset

Fig. 2. Node reordering method of BEARS and its result on the Routing dataset. BEARS reorders nodes so that edges between spokes form a large but sparse block-diagonal submatrix ($\mathbf{H_{11}}$). Diagonal blocks correspond to the connected components detached from the GCC when hubs are removed. Nodes in each block are sorted in the ascending order of their degrees within the component.

According to the definition, computing the Schur complement $\mathbf{S}$ ($n_2 \times n_2$ matrix) of $\mathbf{H_{11}}$ requires $\mathbf{H_{11}^{-1}}$. Instead of directly inverting $\mathbf{H_{11}}$, we LU decompose it into $\mathbf{L_1}$ and $\mathbf{U_1}$, then compute $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ instead (the reason will be explained in Section 3.1.3). Consequently, $\mathbf{S}$ is computed by the following rule:

$$\mathbf{S} = \mathbf{H_{22}} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\mathbf{H_{12}})). \tag{5}$$

3.1.3. LU Decomposition (Lines 7 Through 8). The block elimination method, which will be explained in Section 3.2, requires $\mathbf{H_{11}^{-1}}$ and $\mathbf{S^{-1}}$ to solve Equation (2). Directly inverting $\mathbf{H_{11}}$ and $\mathbf{S}$, however, is inefficient since $\mathbf{S^{-1}}$, as well as each diagonal block of $\mathbf{H_{11}^{-1}}$, is likely to be dense. We avoid this problem by replacing $\mathbf{H_{11}^{-1}}$ with $\mathbf{U_1^{-1}L_1^{-1}}$ as in Equation (5) and replacing $\mathbf{S^{-1}}$ with $\mathbf{U_2^{-1}L_2^{-1}}$, where $\mathbf{L_2}$ and $\mathbf{U_2}$ denote the LU-decomposed matrices of $\mathbf{S}$. To compute $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, and $\mathbf{U_2^{-1}}$ efficiently and make them sparser, we exploit the following observation [Fujiwara et al. 2012a] and lemma.

OBSERVATION 1. *Reordering nodes in ascending order of their degrees speeds up the LU decomposition of an adjacency matrix and makes the inverse of the LU-decomposed matrices sparser.*

LEMMA 3.2. *Suppose that* $\mathbf{A}$ *is a nonsingular block-diagonal matrix that consists of diagonal blocks* $\mathbf{A_1}$ *through* $\mathbf{A_b}$. *Let* $\mathbf{L}$ *and* $\mathbf{U}$ *denote the LU-decomposed matrices of* $\mathbf{A}$, *and let* $\mathbf{L_i}$ *and* $\mathbf{U_i}$ *denote those of* $\mathbf{A_i}$ *for all* i ($1 \le i \le b$). *Then,* $\mathbf{L^{-1}}$ *and* $\mathbf{U^{-1}}$ *are the block-diagonal matrices that consist of* $\mathbf{L_1^{-1}}$ *through* $\mathbf{L_b^{-1}}$ *and* $\mathbf{U_1^{-1}}$ *through* $\mathbf{U_b^{-1}}$, *respectively.*

PROOF. See Appendix A.2. □

These observation and lemma suggest for $\mathbf{H_{11}^{-1}}$ the reordering method that we already used in Section 3.1.1. Since we computed $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ in Section 3.1.2, we only need to process $\mathbf{S}$. We first rearrange the hubs in the ascending order of their degrees within $\mathbf{S}$, which reorders $\mathbf{H_{12}}$, $\mathbf{H_{21}}$, and $\mathbf{H_{22}}$, as well as $\mathbf{S}$. Note that computing $\mathbf{S}$ after reordering the hubs, and reordering the hubs after computing $\mathbf{S}$, produces the same result. After decomposing $\mathbf{S}$ into $\mathbf{L_2}$ and $\mathbf{U_2}$, we compute $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$.

ALGORITHM 2: Query Phase in BearS

Input: seed node: s, precomputed matrices: $\mathbf{L}_1^{-1}$, $\mathbf{U}_1^{-1}$, $\mathbf{L}_2^{-1}$, $\mathbf{U}_2^{-1}$, $\mathbf{H}_{12}$, and $\mathbf{H}_{21}$
Output: relevance scores: $\mathbf{r}$

1: create $\mathbf{q}$ whose sth entry is 1 and the others are 0
2: partition $\mathbf{q}$ into $\mathbf{q_1}$ and $\mathbf{q_2}$
3: compute $\mathbf{r_2} = c(\mathbf{U}_2^{-1}(\mathbf{L}_2^{-1}(\mathbf{q_2} - \mathbf{H}_{21}(\mathbf{U}_1^{-1}(\mathbf{L}_1^{-1}\mathbf{q_1})))))$
4: compute $\mathbf{r_1} = \mathbf{U}_1^{-1}(\mathbf{L}_1^{-1}(c\mathbf{q_1} - \mathbf{H}_{12}\mathbf{r_2}))$
5: create $\mathbf{r}$ by concatenating $\mathbf{r_1}$ and $\mathbf{r_2}$
6: **return r**

3.1.4. Dropping Near-Zero Entries (Line 9, BearS-Approx *Only).* The running time and the memory usage in the query phase of our method largely depend on the number of nonzero entries in $\mathbf{L}_1^{-1}$, $\mathbf{U}_1^{-1}$, $\mathbf{L}_2^{-1}$, $\mathbf{U}_2^{-1}$, $\mathbf{H}_{12}$, and $\mathbf{H}_{21}$. Thus, dropping near-zero entries in them saves time and memory space in the query phase, although it sacrifices little accuracy of $\mathbf{r}$. BearS-Approx drops entries whose absolute value is smaller than the drop tolerance ξ. The effects of different ξ values on accuracy, query time, and memory usage are empirically analyzed in Section 5. Note that contrary to BearS-Approx, BearS-Exact guarantees the exactness of $\mathbf{r}$, which will be proved in Section 3.2, and still outperforms other exact methods in terms of time and space, which will be shown in Section 5.

3.2. Query Phase

In the query phase, BearS computes the RWR score vector $\mathbf{r}$ with respect to a given seed node s by exploiting the results of the preprocessing phase. Algorithm 2 describes the overall procedure of the query phase.

The vector $\mathbf{q} = [\begin{smallmatrix}\mathbf{q_1}\\ \mathbf{q_2}\end{smallmatrix}]$ denotes the length-n starting vector whose entry at the index of the seed node s is 1 and otherwise 0. It is partitioned into the length-n_1 vector $\mathbf{q_1}$ and the length-n_2 vector $\mathbf{q_2}$. The exact RWR score vector $\mathbf{r}$ is computed by the following equation:

$$\mathbf{r} = \begin{bmatrix}\mathbf{r_1}\\ \mathbf{r_2}\end{bmatrix} = \begin{bmatrix}\mathbf{U}_1^{-1}(\mathbf{L}_1^{-1}(c\mathbf{q_1} - \mathbf{H}_{12}\mathbf{r_2}))\\ c(\mathbf{U}_2^{-1}(\mathbf{L}_2^{-1}(\mathbf{q_2} - \mathbf{H}_{21}(\mathbf{U}_1^{-1}(\mathbf{L}_1^{-1}\mathbf{q_1})))))\end{bmatrix}. \quad (6)$$

To prove the correctness of the preceding equation, we use the block elimination method.

Lemma 3.3 (Block elimination [Boyd and Vandenberghe 2009]). *Suppose that a linear equation* $\mathbf{Ax} = \mathbf{b}$ *is partitioned as*

$$\begin{bmatrix}\mathbf{A}_{11} & \mathbf{A}_{12}\\ \mathbf{A}_{21} & \mathbf{A}_{22}\end{bmatrix}\begin{bmatrix}\mathbf{x_1}\\ \mathbf{x_2}\end{bmatrix} = \begin{bmatrix}\mathbf{b_1}\\ \mathbf{b_2}\end{bmatrix},$$

where $\mathbf{A}_{11}$ *and* $\mathbf{A}_{22}$ *are square matrices. If the submatrix* $\mathbf{A}_{11}$ *is invertible, and* $\mathbf{S}$ *is the Schur complement of the submatrix* $\mathbf{A}_{11}$ *in* $\mathbf{A}$,

$$\mathbf{x} = \begin{bmatrix}\mathbf{x_1}\\ \mathbf{x_2}\end{bmatrix} = \begin{bmatrix}\mathbf{A}_{11}^{-1}(\mathbf{b_1} - \mathbf{A}_{12}\mathbf{x_2})\\ \mathbf{S}^{-1}(\mathbf{b_2} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b_1})\end{bmatrix}. \quad (7)$$

Theorem 3.4 (The correctness of BearS-Exact). *The* $\mathbf{r}$ *in Equation* (6) *is equal to the* $\mathbf{r}$ *in Equation* (2).

Proof. $\mathbf{H}_{11}$ is invertible because its transpose is a strictly diagonally dominant matrix for $0 < c < 1$ [Banerjee and Roy 2014]. Thus, by Lemma 3.3, the following holds

Table II. Time Complexity of Each Step in BEARS

Line	Task	Time Complexity
	Preprocessing Phase (Algorithm 1)	
2	run SlashBurn	$O(T(m+n\log n))$ [Kang and Faloutsos 2011]
3	reorder $\mathbf{H}$	$O(m+n+\sum_{i=1}^{b} n_{1i}\log n_{1i})$
5	compute $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$	$O(\sum_{i=1}^{b} n_{1i}^3)$
6	compute $\mathbf{S}$	$O((n_2\sum_{i=1}^{b} n_{1i}^2)+\min(n_1n_2^2, n_2m))$
8	compute $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$	$O(n_2^3)$
9	drop near-zero entries	$O((\sum_{i=1}^{b} n_{1i}^2)+n_2^2+min(n_1n_2, m))$
Total	$O(T(m+n\log n)+(\sum_{i=1}^{b} n_{1i}^3)+(n_2\sum_{i=1}^{b} n_{1i}^2)+n_2^3+\min(n_2^2n_1, n_2m))$	
	Query Phase (Algorithm 2)	
3	compute $\mathbf{r_2}$	$O((\sum_{i=1}^{b} n_{1i}^2)+n_2^2+\min(n_1n_2, m))$
4	compute $\mathbf{r_1}$	$O((\sum_{i=1}^{b} n_{1i}^2)+\min(n_1n_2, m))$
Total	$O((\sum_{i=1}^{b} n_{1i}^2)+n_2^2+\min(n_1n_2, m))$	

Table III. Maximum Number of Nonzero Entries in the Precomputed Matrices

Matrix	Max Nonzeros
$\mathbf{H_{12}}$ & $\mathbf{H_{21}}$	$O(\min(n_1n_2, m))$
$\mathbf{L_1^{-1}}$ & $\mathbf{U_1^{-1}}$	$O(\sum_{i=1}^{b} n_{1i}^2)$
$\mathbf{L_2^{-1}}$ & $\mathbf{U_2^{-1}}$	$O(n_2^2)$

for Equation (2):

$$\mathbf{r} = \begin{bmatrix} \mathbf{r_1} \\ \mathbf{r_2} \end{bmatrix} = \begin{bmatrix} \mathbf{H_{11}^{-1}}(c\mathbf{q_1} - \mathbf{H_{12}r_2}) \\ \mathbf{S}^{-1}(c\mathbf{q_2} - \mathbf{H_{21}H_{11}^{-1}}(c\mathbf{q_1})) \end{bmatrix}.$$

Equation (6) only replaces $\mathbf{H_{11}^{-1}}$ with $\mathbf{U_1^{-1}L_1^{-1}}$ and $\mathbf{S}^{-1}$ with $\mathbf{U_2^{-1}L_2^{-1}}$ where $\mathbf{H_{11}} = \mathbf{L_1U_1}$ and $\mathbf{S} = \mathbf{L_2U_2}$. □

3.3. Complexity Analysis

In this section, we analyze the time and space complexity of BEARS. We assume that all matrices considered are saved in a sparse format, such as the compressed sparse column format [Press 2007], which stores only nonzero entries, and that all matrix operations exploit such sparsity by only considering nonzero entries. We also assume that the number of edges is greater than that of nodes (i.e., $m > n$) for simplicity. The maximum number of nonzero entries in each precomputed matrix is summarized in TableIII, where b denotes the number of diagonal blocks in $\mathbf{H_{11}}$ and n_{1i} denotes the number of nodes in the ith diagonal block. The maximum numbers of nonzero entries in $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ depend on the size of diagonal blocks because the LU-decomposed matrices and inverse of a block-diagonal matrix are also block-diagonal matrices with the same block sizes (see Lemma 3.2).

3.3.1. Time Complexity. The time complexity of each step of BEARS is summarized in Table II. In this section, we provide proofs for the time complexity of the proposed method with the following lemma.

LEMMA 3.5 (SPARSE MATRIX MULTIPLICATION). *Suppose that* $\mathbf{A}$ *and* $\mathbf{B}$ *are* $p \times q$ *and* $q \times r$ *matrices, respectively, and* $\mathbf{A}$ *has* $|\mathbf{A}|$ $(> p, q)$ *nonzero entries. Calculating* $\mathbf{C} = \mathbf{AB}$ *using sparse matrix multiplication takes* $O(|\mathbf{A}|r)$.

PROOF. Each nonzero entry in $\mathbf{A}$ is multiplied and then added up to r times. □

THEOREM 3.6. *The preprocessing phase in* BEARS *takes* $O(T(m+n\log n)+(\sum_{i=1}^{b} n_{1i}^3)+(n_2\sum_{i=1}^{b} n_{1i}^2)+n_2^3+\min(n_2^2 n_1, n_2 m))$.

PROOF. See Table II. To compute $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$, it takes $O(\sum_{i=1}^{b} n_{1i}^3)$ [Boyd and Vandenberghe 2009]. To compute $\mathbf{S}$, it takes $O(n_2\sum_{i=1}^{b} n_{1i}^2 + \min(n_1 n_2^2, n_2 m))$ because it takes $O(n_2\sum_{i=1}^{b} n_{1i}^2)$ to compute $\mathbf{R_1} = \mathbf{L_1^{-1}H_{12}}$, $O(n_2\sum_{i=1}^{b} n_{1i}^2)$ to compute $\mathbf{R_2} = \mathbf{U_1^{-1}R_1}$, $O(\min(n_1 n_2^2, n_2 m))$ to compute $\mathbf{R_3} = \mathbf{H_{21}R_2}$ by Lemma 3.5, and $O(n_2^2)$ to compute $\mathbf{S} = \mathbf{H_{22}} - \mathbf{R_3}$. It takes $O(n_2^3)$ to compute $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$. □

THEOREM 3.7. *The query phase in* BEARS *takes* $O((\sum_{i=1}^{b} n_{1i}^2) + n_2^2 + \min(n_1 n_2, m))$.

PROOF. Apply Lemma 3.5 and the results in Table III to each step in $\mathbf{r_2} = c(\mathbf{U_2^{-1}}(\mathbf{L_2^{-1}}(\mathbf{q_2} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\mathbf{q_1})))))$ and $\mathbf{r_1} = \mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}(c\mathbf{q_1} - \mathbf{H_{12}r_2}))$. □

In real-world graphs, $\sum_{i=1}^{b} n_{1i}^2$ in the preceding results can be replaced by m since the number of nonzero entries in $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ is closer to $O(m)$ than $O(\sum_{i=1}^{b} n_{1i}^2)$, as seen later in Table VI.

3.3.2. Space Complexity.

THEOREM 3.8. BEARS *requires* $O((\sum_{i=1}^{b} n_{1i}^2) + \min(n_1 n_2, m) + n_2^2)$ *memory space for precomputed matrices:* $\mathbf{H_{12}}$, $\mathbf{H_{21}}$, $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, *and* $\mathbf{U_2^{-1}}$.

PROOF. See Table III. □

For the same reason as in the time complexity, $\sum_{i=1}^{b} n_{1i}^2$ in the preceding result can be replaced by m in real-world graphs.

Theorems 3.6, 3.7, and 3.8 imply that BEARS works efficiently when the given graph is divided into small pieces (small $\sum_{i=1}^{b} n_{1i}^2$ and $\sum_{i=1}^{b} n_{1i}^3$) by removing a small number of hubs (small n_2), which is true in many real-world graphs [Albert et al. 2000; Kang and Faloutsos 2011].

3.4. More Applications

Our BEAR method is easily applicable to various RWR variants, as BEAR does not assume any unique property of RWR contrary to other methods [Fujiwara et al. 2012a; Wu et al. 2014]. In this section, we show how BEAR can be applied to three of these variants. Furthermore, we present that BEAR also can be applied to other graph similarities, such as FaBP [Koutra et al. 2011, 2013].

Personalized Page-Rank. As explained in Section 2.1, PPR selects a restart node according to given probability distribution. PPR can be computed by replacing $\mathbf{q}$ in Algorithm 2 with the probability distribution.

Effective importance. EI [Bogdanov and Singh 2013] is the degree-normalized version of RWR. It captures the local community structure and adjusts RWR's preference toward high-degree nodes. We can compute EI by dividing each entry of $\mathbf{r}$ in Algorithm 2 by the degree of the corresponding node.

RWR with normalized graph Laplacian. Instead of row-normalized adjacency matrix, Tong et al. [2008] use the normalized graph Laplacian. It outputs symmetric relevance

ALGORITHM 3: Update Phase in BEARD

Input: matrices: $\mathbf{D}$, $\mathbf{H}$, $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{S}$, and an edge: (u, v, w)
Output: updated matrices: $\widehat{\mathbf{D}}$, $\widehat{\mathbf{H}}$, $\widehat{\mathbf{L}}_1^{-1}$, $\widehat{\mathbf{U}}_1^{-1}$, $\widehat{\mathbf{S}}$, $\widehat{\mathbf{L}}_2$, and $\widehat{\mathbf{U}}_2$

1: update $\mathbf{D}$ and $\mathbf{H}$ into $\widehat{\mathbf{D}}$ and $\widehat{\mathbf{H}}$ by Algorithm 4
2: update $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ into $\widehat{\mathbf{L}}_1^{-1}$ and $\widehat{\mathbf{U}}_1^{-1}$ by Algorithm 5
3: update $\mathbf{S}$ into $\widehat{\mathbf{S}}$ by Algorithm 6
4: decompose $\widehat{\mathbf{S}}$ into $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ by LU decomposition
5: **return** $\widehat{\mathbf{D}}$, $\widehat{\mathbf{H}}$, $\widehat{\mathbf{L}}_1^{-1}$, $\widehat{\mathbf{U}}_1^{-1}$, $\widehat{\mathbf{S}}$, $\widehat{\mathbf{L}}_2$, and $\widehat{\mathbf{U}}_2$

scores for undirected graphs, which are desirable for some applications. This score can be computed by replacing $\tilde{\mathbf{A}}$ in Algorithm 1 with $\mathbf{D^{-1/2}AD^{-1/2}}$, where $\mathbf{A}$ is an unnormalized adjacency matrix and $\mathbf{D}$ is a diagonal matrix whose (i, i)th entry is the degree of ith node.

Fast belief propagation. FaBP [Koutra et al. 2011] is a fast algorithm for approximate belief propagation. The linear system that FaBP solves is expressed as $[\mathbf{I} + a\mathbf{D} - c'\mathbf{A}]\mathbf{b}_h = \boldsymbol{\phi}_h$, where $\mathbf{b}_h$ is an approximation of the beliefs, $\boldsymbol{\phi}_h$ is a prior belief vector, $a = 4h_h^2/(1 - 4h_h^2)$, $c = 2h_h/(1 - 4h_h^2)$, and h_h is a homophily factor. Note that the matrix $\mathbf{H} = [\mathbf{I} + a\mathbf{D} - c'\mathbf{A}]$ is invertible if $\frac{1}{2(1-max(D_{ii}))} < h_h < 0$ and $0 < h_h < \frac{1}{2(max(D_{ii})-1)}$ since $\mathbf{H}$ becomes diagonally dominant as proved in Lemma 8.1 in Appendix A.3. Under the condition, BEAR computes the solution of the FaBP equation.

4. PROPOSED METHOD FOR DYNAMIC GRAPHS

In this section, we propose BEARD to apply BEAR to dynamic graphs. When edges are added to or removed from a graph, BEARD updates the preprocessed matrices of BEARS based on the following observation:

—When a few edges are inserted or deleted, only small parts of the preprocessed matrices need to be changed. In other words, most parts of the preprocessed matrices remain the same.

For example, suppose that one edge is inserted into a graph as shown in the left of Figure 3(a). In this case, the inserted edge (the red dot) results in the change of one column (the red line) in the normalized adjacency matrix of the graph. The modified column leads to the small change of the preprocessed matrices because the preprocessed matrices are computed from the normalized adjacency matrix. In this manner, BEARD analyzes the changes of the normalized adjacency matrix and updates the preprocessed matrices accordingly.

As the dynamic graph model, we choose the edge weighted dynamic graph [Harary and Gupta 1997], in which edges are added with a weight, deleted from the graph, or the weights of edges are modified while the set of nodes is fixed. For simplicity, we refer the insertion, the deletion, and the weight change of an edge to an *edge modification*. One edge modification, (u, v, w), means that the weight of an edge from node u to v is replaced with the weight w. For instance, if we set the weight of an edge (u, v) to 0, then this modification is represented as $(u, v, 0)$ and indicates the deletion of the edge. Note that we do not consider the cases when nodes are added or removed; it is a challenging open problem since addition or removal of nodes changes the dimension of the matrices, which makes updating the preprocessed matrices difficult.

BEARD comprises the update phase and the query phase. In the update phase, BEARD updates the preprocessed matrices for a given edge modification (u, v, w). In the query phase, BEARD computes RWR scores with respect to a given seed node s using the

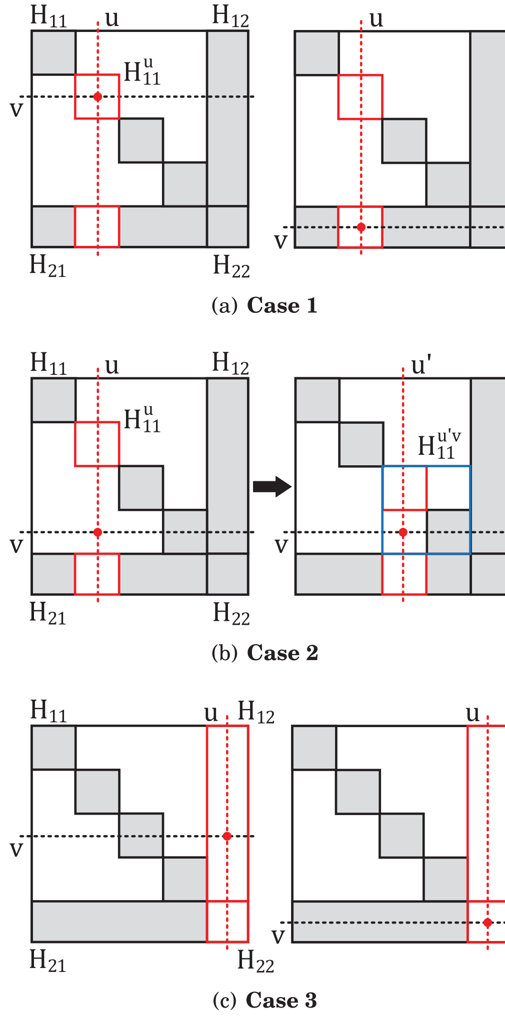


Fig. 3. Cases of changes in $\mathbf{H}$ after one edge modification. A red dot indicates the position of a modified edge, and a red box indicates the part of $\mathbf{H}$ modified by the edge modification. (a) Case 1, where the modification occurs in one block of $\mathbf{H}_{11}$ (left) or in $\mathbf{H}_{21}$(right). (b) Case 2, where an edge is added between two blocks. The blocks may be adjacent or not. If they are not adjacent, we reorder $\mathbf{H}_{11}$ (and thus $\mathbf{H}_{21}$) and merge them into one block (e.g., the blue box on the right) so that $\mathbf{H}_{11}$ remains block diagonal. (c) Case 3, where the edge modification occurs in $\mathbf{H}_{12}$ (left) or in $\mathbf{H}_{22}$ (right).

updated matrices. The update and query phases of BEARD are shown in Algorithms 3 and 7, respectively.

4.1. Update Phase

In the update phase, for one edge modification, BEARD updates the preprocessed matrices $\mathbf{D}$, $\mathbf{H}$, $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, and $\mathbf{S}$, which are computed in BEARS. $\mathbf{D}$ is a diagonal matrix with $\mathbf{D}_{ii} = \sum_j \mathbf{A}_{ij}$, $\mathbf{H} = \mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}$, $\mathbf{H}_{11} = \mathbf{L_1}\mathbf{U_1}$ by LU decomposition, and $\mathbf{S}$ is the Schur complement of $\mathbf{H}_{11}$. We use $\widehat{\mathbf{A}}$ to denote the updated matrix of $\mathbf{A}$ and $\Delta\mathbf{A}$ to denote the difference $\widehat{\mathbf{A}} - \mathbf{A}$ after one edge modification.

ALGORITHM 4: Updating $\mathbf{D}$ and $\mathbf{H}$

Input: restart probability: c, matrices : $\mathbf{D}$ and $\mathbf{H}$, and an edge: (u, v, w)
Output: updated matrices: $\widehat{\mathbf{D}}$ and $\widehat{\mathbf{H}}$

```
1: initialize ΔD and ΔH
2: ΔD_uu = w + H_vu D_uu/(1 − c)
3: for v′ ∈ N(u) do
4:     ΔH_v′u = −H_v′u ΔD_uu/(D_uu + ΔD_uu)
5: end for
6: ΔH_vu = −H_vu − (1 − c)w/D̂_uu
7: D̂ = D + ΔD
8: Ĥ = H + ΔH
9: return D̂ and Ĥ
```

After one edge is inserted or deleted, $\mathbf{D}$ and $\mathbf{H}$ are changed because the degree of the source node of the edge is changed. If $\mathbf{H}_{11}$ is changed, then $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ need to be updated because the matrices are computed from $\mathbf{H}_{11}$. In addition, the change of $\mathbf{H}$ incurs the update of $\mathbf{S}$ since $\mathbf{S}$ is computed from submatrices of $\mathbf{H}$. Finally, the update of $\mathbf{S}$ changes $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$ where $\mathbf{S}$ is LU decomposed into $\mathbf{L_2}$ and $\mathbf{U_2}$.

In this sense, BEARD must update $\mathbf{D}$, $\mathbf{H}$, $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, and $\mathbf{U_2^{-1}}$ when the graph changes. However, BEARD does not update $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$. Instead, BEARD updates $\mathbf{S}$ into $\widehat{\mathbf{S}}$ and computes $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ through LU decomposition of $\widehat{\mathbf{S}}$. The reason is that to update $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$, we need to LU decompose $\widehat{\mathbf{S}}$ into $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$, and then invert the new LU factors, but the task for inverting the matrices takes a long time. To decrease the update time, BEARD avoids inverting $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$, and computes a query using $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$, which will be explained in detail in Section 4.2.

The update phase is composed of four steps: (1) updating $\mathbf{D}$ and $\mathbf{H}$, (2) updating $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$, (3) updating $\mathbf{S}$, and (4) computing $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ from $\widehat{\mathbf{S}}$, where $\widehat{\mathbf{S}}$ is the updated matrix of $\mathbf{S}$ and $\widehat{\mathbf{S}} = \widehat{\mathbf{L}}_2\widehat{\mathbf{U}}_2$. Details of the update phase (Algorithm 3) are explained in the following sections. Note that the output matrices of Algorithm 3 are used as the input matrices of the next edge modification.

4.1.1. Updating $\mathbf{D}$ *and* $\mathbf{H}$. In this part, we update $\mathbf{D}$ and $\mathbf{H}$ after computing $\Delta\mathbf{D}$ and $\Delta\mathbf{H}$ for an edge modification (u, v, w). When the edge (u, v) is modified, $\mathbf{D}_{uu}$, the (u, u)th entry of $\mathbf{D}$, needs to be updated because the degree of the uth node is changed. In turn, the change of $\mathbf{D}_{uu}$ leads to the update of the uth column of $\mathbf{H}$ according to the definition of $\mathbf{H}$.

If the weight of an edge (u, v) is changed to w, then the (u, v)th entry of $\mathbf{A}$, $\mathbf{A}_{uv}$ becomes w, and $\mathbf{D}_{uu}$ is changed into $\widehat{\mathbf{D}}_{uu}$ as follows:

$$\widehat{\mathbf{D}}_{uu} = \mathbf{D}_{uu} - \mathbf{A}_{uv} + w.$$

According to the preceding equation, $\Delta\mathbf{D}$ is $-\mathbf{A}_{uv} + w$ on (u, u)th entry and zeros on the others. The change of $\mathbf{D}$ indicates that the uth row of the normalized adjacency matrix $\tilde{\mathbf{A}}$ needs to be updated because only the (u, u)th entry of $\mathbf{D}$ changes. Equally, the uth column of $\mathbf{H}$ must be changed because $\tilde{\mathbf{A}}$ is transposed as shown in Equation (2). To update the values on the uth column of $\mathbf{H}$, we rewrite $\mathbf{H}$ as follows:

$$\mathbf{H}_{ij} = \begin{cases} -(1-c)\dfrac{\mathbf{A}^{\top}_{ij}}{\mathbf{D}_{jj}}, & \text{if } i \neq j \\ 1, & \text{otherwise,} \end{cases}$$

where $\mathbf{H}_{ij}$ is the (i, j)th entry of $\mathbf{H}$. Hence, $\mathbf{A}_{uv}$ is $-\mathbf{D}_{uu} \times \mathbf{H}_{vu}/(1-c)$. After the edge modification, each nonzero element in the uth column of $\widehat{\mathbf{H}}$ is updated as follows:

$$\widehat{\mathbf{H}}_{iu} = \begin{cases} -(1-c)\dfrac{w}{\widehat{\mathbf{D}}_{uu}} & \text{if } i = v \\ \mathbf{H}_{iu}\dfrac{\mathbf{D}_{uu}}{\widehat{\mathbf{D}}_{uu}} & \text{else if } i \in N(u), \end{cases}$$

where $N(u)$ is a set of out-neighbors of node u. We derive the uth column of $\Delta\mathbf{H}$ from the relation $\widehat{\mathbf{H}} = \mathbf{H} + \Delta\mathbf{H}$ as follows:

$$\Delta\mathbf{H}_{iu} = \begin{cases} -\mathbf{H}_{iu} - (1-c)\dfrac{w}{\widehat{\mathbf{D}}_{uu}} & \text{if } i = v \\ -\dfrac{\Delta\mathbf{D}_{uu}}{\widehat{\mathbf{D}}_{uu}}\mathbf{H}_{iu} & \text{else if } i \in N(u). \end{cases}$$

The process for computing $\Delta\mathbf{D}$ and $\Delta\mathbf{H}$ is shown in Algorithm 4. After obtaining $\Delta\mathbf{D}$ and $\Delta\mathbf{H}$, we update $\mathbf{D}$ into $\widehat{\mathbf{D}} = \mathbf{D} + \Delta\mathbf{D}$ and $\mathbf{H}$ into $\widehat{\mathbf{H}} = \mathbf{H} + \Delta\mathbf{H}$. Note that an updated edge changes only one column of $\mathbf{H}$. Thus, $\Delta\mathbf{H}$ contains nonzero elements only in the uth column.

4.1.2. Case Analysis. Depending on $\Delta\mathbf{H}$, which results from an edge modification (u, v, w), we adopt different approaches for updating the matrices given as input. For the purpose, we classify $\Delta\mathbf{H}$ into three cases and analyze each case in turn. Before describing the analysis, we define a symbol to represent a block in $\mathbf{H}_{11}$ for clarity.

Definition 4.1 (*A Block of Node u*). Suppose that $\mathbf{A}_{11}$ is a block-diagonal matrix and i_u is the index of the block to which node u belongs. Then, $\mathbf{A}_{11}^{i_u}$ indicates the block of the index i_u.

Case 1 *(Figure 3(a))*. An edge modification occurs within a block $\mathbf{H}_{11}^{i_u} (= \mathbf{H}_{11}^{i_v})$ or within $\mathbf{H}_{21}$. In this case, the uth columns of $\mathbf{H}_{11}^{i_u}$ and $\mathbf{H}_{21}$ are changed. The others, $\mathbf{H}_{12}$ and $\mathbf{H}_{22}$, are not changed.

Case 2 *(Figure 3(b))*. An edge is added between a node in $\mathbf{H}_{11}^{i_u}$ and a node in $\mathbf{H}_{11}^{i_v}$. If these two blocks are not adjacent, we reorder $\mathbf{H}_{11}$ so that they become adjacent and the block-diagonal structure of $\mathbf{H}_{11}$ is preserved. As in Case 1, only $\mathbf{H}_{11}$ and $\mathbf{H}_{21}$ are modified.

Case 3 *(Figure 3(c))*. An edge modification occurs within $\mathbf{H}_{12}$ or within $\mathbf{H}_{22}$. The uth columns of $\mathbf{H}_{12}$ and $\mathbf{H}_{22}$ are changed, whereas $\mathbf{H}_{11}$ and $\mathbf{H}_{21}$ are not changed.

4.1.3. Updating $\mathbf{L}_1^{-1}$ *and* $\mathbf{U}_1^{-1}$. In this part, we update $\mathbf{L}_1^{-1}$ and $\mathbf{U}_1^{-1}$ to $\widehat{\mathbf{L}}_1^{-1}$ and $\widehat{\mathbf{U}}_1^{-1}$ based on $\widehat{\mathbf{H}}$, which is the updated matrix of $\mathbf{H}$. If any part of $\mathbf{H}_{11}$ is changed (Cases 1 and 2), $\mathbf{L}_1^{-1}$ and $\mathbf{U}_1^{-1}$ should be updated because they are computed from $\mathbf{H}_{11}$. On the other hand, if $\mathbf{H}_{11}$ is not modified (Case 3), the factors do not need to be updated. Hence, we focus on Cases 1 and 2 when $\mathbf{L}_1^{-1}$ and $\mathbf{U}_1^{-1}$ are updated in this step.

Note that $\mathbf{L}_1^{-1}$ and $\mathbf{U}_1^{-1}$, as well as $\mathbf{H}_{11}$, are block-diagonal matrices by Lemma 3.2. By one edge modification, the uth column of $\mathbf{H}_{11}$ is changed; thus, only one block in $\mathbf{H}_{11}$ is modified, and other blocks are not changed. Therefore, we update $\mathbf{L}_1^{-1}$ and $\mathbf{U}_1^{-1}$ based on the modified block case by case.

Case 1. Since $\mathbf{H}_{11}$ is a block-diagonal matrix, and only one block $\mathbf{H}_{11}^{i_u}$ is changed by the edge modification as seen in Figure 3(a), we keep the unchanged blocks in $\mathbf{L}_1^{-1}$ and $\mathbf{U}_1^{-1}$, and we use them to initialize $\widehat{\mathbf{L}}_1^{-1}$ and $\widehat{\mathbf{U}}_1^{-1}$. We replace the only modified block. As described in Algorithm 5, we extract one block $\widehat{\mathbf{H}}_{11}^{i_u}$ from $\widehat{\mathbf{H}}$, LU decompose it into $\overline{\mathbf{L}}_1^{i_u}$

ALGORITHM 5: Updating $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$

Input: matrices : $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, and $\widehat{\mathbf{H}}$, an edge: (u, v, w)
Output: updated matrices: $\widehat{\mathbf{L}}_1^{-1}$ and $\widehat{\mathbf{U}}_1^{-1}$

1: initialize $\widehat{\mathbf{L}}_1^{-1} \leftarrow \mathbf{L_1^{-1}}$, and $\widehat{\mathbf{U}}_1^{-1} \leftarrow \mathbf{U_1^{-1}}$
2: **if Case 3 then**
3: /*no need to update $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$*/
return $\widehat{\mathbf{L}}_1^{-1}$ and $\widehat{\mathbf{U}}_1^{-1}$
4: **end if**
5: **if Case 1 then**
6: $\widehat{\mathbf{H}}_{11}^{i_{uv}} \leftarrow \widehat{\mathbf{H}}_{11}^{i_u}$
7: **else if Case 2 then**
8: **if** $\widehat{\mathbf{H}}_{11}^{i_u}$ is not adjacent to $\widehat{\mathbf{H}}_{11}^{i_v}$ **then**
9: reorder $\widehat{\mathbf{H}}$, $\widehat{\mathbf{L}}_1^{-1}$ and $\widehat{\mathbf{U}}_1^{-1}$ for $\widehat{\mathbf{H}}_{11}^{i_u}$ and $\widehat{\mathbf{H}}_{11}^{i_v}$ to be adjacent
10: **end if**
11: merge $\widehat{\mathbf{H}}_{11}^{i_u}$ and $\widehat{\mathbf{H}}_{11}^{i_v}$ into one block $\widehat{\mathbf{H}}_{11}^{i_{uv}}$
12: **end if**
13: decompose $\widehat{\mathbf{H}}_{11}^{i_{uv}}$ into $\overline{\mathbf{L}}_1^{i_{uv}}$ and $\overline{\mathbf{U}}_1^{i_{uv}}$, and compute $(\overline{\mathbf{L}}_1^{i_{uv}})^{-1}$ and $(\overline{\mathbf{U}}_1^{i_{uv}})^{-1}$
14: replace $(\widehat{\mathbf{L}}_1^{i_{uv}})^{-1}$ and $(\widehat{\mathbf{U}}_1^{i_{uv}})^{-1}$ with $(\overline{\mathbf{L}}_1^{i_{uv}})^{-1}$ and $(\overline{\mathbf{U}}_1^{i_{uv}})^{-1}$, respectively.
15: **return** $\widehat{\mathbf{L}}_1^{-1}$ and $\widehat{\mathbf{U}}_1^{-1}$

and $\overline{\mathbf{U}}_1^{i_u}$, and invert them. Then, we replace $(\widehat{\mathbf{L}}_1^{i_u})^{-1}$ and $(\widehat{\mathbf{U}}_1^{i_u})^{-1}$ with $(\overline{\mathbf{L}}_1^{i_u})^{-1}$ and $(\overline{\mathbf{U}}_1^{i_u})^{-1}$, respectively.

Case 2. As illustrated in Figure 3(b), if $\widehat{\mathbf{H}}_{11}^{i_u}$ is not adjacent to $\widehat{\mathbf{H}}_{11}^{i_v}$, we reorder $\widehat{\mathbf{H}}_{11}$ for them to be adjacent. When $\widehat{\mathbf{H}}_{11}$ is reordered, $\mathbf{D}$, $\widehat{\mathbf{H}}_{12}$, $\widehat{\mathbf{H}}_{21}$, $\widehat{\mathbf{L}}_1^{-1}$, and $\widehat{\mathbf{U}}_1^{-1}$ also should be reordered in the same way for consistency. Then, we merge two blocks $\widehat{\mathbf{H}}_{11}^{i_u}$ and $\widehat{\mathbf{H}}_{11}^{i_v}$ into one block $\widehat{\mathbf{H}}_{11}^{i_{uv}}$ that denotes the block to which both u and v belong. Note that if we do not reorder $\mathbf{H}_{11}$ and merge the blocks, the LU factors of $\widehat{\mathbf{H}}_{11}$ become dense, and the performance degrades in the update and query phases.

4.1.4. Updating $\mathbf{S}$. In this section, we compute $\Delta\mathbf{S}$ and update $\mathbf{S}$ to $\widehat{\mathbf{S}}$, where $\mathbf{S}$ is the Schur complement of $\mathbf{H}_{11}$. According to Equation (5), if at least one of the submatrices in $\mathbf{H}$ is changed, $\mathbf{S}$ is also changed. Hence, every case of edge modification incurs the change of $\mathbf{S}$. To compute $\Delta\mathbf{S}$, we reformulate $\mathbf{S}$ using blocks in $\mathbf{H}_{11}$ and define *block contribution* to $\mathbf{S}$.

LEMMA 4.2. *Suppose that* $\mathbf{A}_{11}$ *is a block-diagonal matrix, and* $\mathbf{A}_{12}^i$*,* $\mathbf{A}_{21}^i$*, and* $\mathbf{A}_{11}^i$ *denote the parts of* $\mathbf{A}_{12}$*,* $\mathbf{A}_{21}$*, and* $\mathbf{A}_{11}$ *corresponding to the block of index i, respectively. Then, the Schur complement* $\mathbf{S}$ *of* $\mathbf{A}_{11}$ *can be expressed as follows:*

$$\mathbf{S} = \mathbf{A}_{22} - \sum_{i=1}^{b} \mathbf{A}_{21}^i (\mathbf{A}_{11}^i)^{-1} \mathbf{A}_{12}^i. \tag{8}$$

PROOF. Since $\mathbf{A}_{11}$ is a block-diagonal matrix, $\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ can be represented as follows:

$$\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12} = \begin{bmatrix}\mathbf{A}_{21}^1 & \dots & \mathbf{A}_{21}^b\end{bmatrix} \begin{bmatrix} (\mathbf{A}_{11}^1)^{-1} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & (\mathbf{A}_{11}^b)^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{12}^1 \\ \vdots \\ \mathbf{A}_{12}^b \end{bmatrix}.$$

Then, the term and the Schur complement of $\mathbf{A}_{11}$ are expressed as Equation (8). □

ALGORITHM 6: Updating $\mathbf{S}$

Input: matrices: $\mathbf{H}$, $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\widehat{\mathbf{H}}$, $\widehat{\mathbf{L}}_1^{-1}$, $\widehat{\mathbf{U}}_1^{-1}$, an edge: (u, v, w)
Output: updated matrix: $\widehat{\mathbf{S}}$

1: **if Case 1 then**
2: compute $\Delta\mathbf{S} = -\Gamma(\mathbf{H}, i_u) + \Gamma(\widehat{\mathbf{H}}, i_u)$
3: **else if Case 2 then**
4: compute $\Delta\mathbf{S} = -\Gamma(\mathbf{H}, i_u) - \Gamma(\mathbf{H}, i_v) + \Gamma(\widehat{\mathbf{H}}, i_{uv})$
5: **else if Case 3 then**
6: compute $\Delta\mathbf{S} = \Delta\mathbf{H}_{22} - \mathbf{H}_{21}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\Delta\mathbf{H}_{12}))$
7: **end if**
8: $\widehat{\mathbf{S}} = \mathbf{S} + \Delta\mathbf{S}$
9: **return** $\widehat{\mathbf{S}}$

Definition 4.3 (*Block Contribution to Schur Complement*). Suppose that $\mathbf{A}_{11}$ is a block-diagonal matrix. Given a block index i, let $\mathbf{A}_{12}^i$, $\mathbf{A}_{21}^i$, and $\mathbf{A}_{11}^i$ denote the parts of $\mathbf{A}_{12}$, $\mathbf{A}_{21}$, and $\mathbf{A}_{11}$, respectively, corresponding to the block i. Then, the contribution of block i to the Schur complement $\mathbf{S}$ is defined by

$$\Gamma(\mathbf{A}, i) = -\mathbf{A}_{21}^i\left(\mathbf{A}_{11}^i\right)^{-1}\mathbf{A}_{12}^i.$$

If $\mathbf{A}_{11}$ is LU decomposed into $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$, then the block contribution can be rewritten using the inverse of the factors as follows:

$$\Gamma(\mathbf{A}, i) = -\mathbf{A}_{21}^i\left(\left({\mathbf{U}_{11}}^i\right)^{-1}\left(\left({\mathbf{L}_{11}}^i\right)^{-1}\mathbf{A}_{12}^i\right)\right).$$

Using Lemma 4.2 and Definition 4.3, the Schur complement of $\mathbf{H}_{11}$ is represented by the following equation:

$$\mathbf{S} = \mathbf{H}_{22} + \sum_{i=1}^{b}\Gamma(\mathbf{H}, i).$$

As mentioned in Section 4.1.2, at most two blocks in $\mathbf{H}_{11}$ are replaced by the edge modification. The key idea for computing $\Delta\mathbf{S}$ is to extract the block contributions of old blocks from $\mathbf{S}$ and to add those of newly replaced blocks.

Case 1. The modified block in $\mathbf{H}_{11}$ is only $\mathbf{H}_{11}^{i_u}$. Therefore, we extract the contribution of the replaced block $\mathbf{H}_{11}^{i_u}$ from $\mathbf{S}$ and add that of $\widehat{\mathbf{H}}_{11}^{i_u}$ into $\widehat{\mathbf{S}}$ as follows:

$$\Delta\mathbf{S} = -\Gamma(\mathbf{H}, i_u) + \Gamma(\widehat{\mathbf{H}}, i_u). \tag{9}$$

Case 2. In the previous step, we replaced $\mathbf{H}_{11}^{i_u}$ and $\mathbf{H}_{11}^{i_v}$ with the merged block, $\widehat{\mathbf{H}}_{11}^{i_{uv}}$. Thus, we extract the contribution of $\mathbf{H}_{11}^{i_u}$ and $\mathbf{H}_{11}^{i_v}$, and then we add that of $\widehat{\mathbf{H}}_{11}^{i_{uv}}$ into $\widehat{\mathbf{S}}$ as follows:

$$\Delta\mathbf{S} = -\Gamma(\mathbf{H}, i_u) - \Gamma(\mathbf{H}, i_v) + \Gamma(\widehat{\mathbf{H}}, i_{uv}). \tag{10}$$

Case 3. No block in $\mathbf{H}_{11}$ is changed, but elements of the uth columns of $\mathbf{H}_{12}$ and $\mathbf{H}_{22}$ are modified. In this case, we compute $\Delta\mathbf{S}$ using fact that $\widehat{\mathbf{H}} = \mathbf{H} + \Delta\mathbf{H}$. Note that $\widehat{\mathbf{H}}_{21} = \mathbf{H}_{21}$ and $\widehat{\mathbf{H}}_{11} = \mathbf{H}_{11}$ in this case:

$$\begin{aligned}
\widehat{\mathbf{S}} &= \widehat{\mathbf{H}}_{22} - \widehat{\mathbf{H}}_{21}\widehat{\mathbf{H}}_{11}^{-1}\widehat{\mathbf{H}}_{12} \\
&= \mathbf{H}_{22} + \Delta\mathbf{H}_{22} - \mathbf{H}_{21}\mathbf{H}_{11}^{-1}(\mathbf{H}_{12} + \Delta\mathbf{H}_{12}) \\
&= \mathbf{H}_{22} - \mathbf{H}_{21}\mathbf{H}_{11}^{-1}\mathbf{H}_{12} + \Delta\mathbf{H}_{22} - \mathbf{H}_{21}\mathbf{H}_{11}^{-1}\Delta\mathbf{H}_{12} \\
&= \mathbf{S} + \Delta\mathbf{H}_{22} - \mathbf{H}_{21}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\Delta\mathbf{H}_{12})).
\end{aligned}$$

ALGORITHM 7: Query Phase in BEARD

Input: restart probability: c, matrices: $\widehat{\mathbf{H}}$, $\widehat{\mathbf{L}}_1^{-1}$, $\widehat{\mathbf{U}}_1^{-1}$, $\widehat{\mathbf{L}}_2$, and $\widehat{\mathbf{U}}_2$, seed node: s
Output: relevance scores: $\mathbf{r}$

1: create $\mathbf{q}$ whose sth entry is 1 and the others are 0
2: partition $\mathbf{q}$ into $\mathbf{q_1}$ and $\mathbf{q_2}$
3: compute $\mathbf{r_2} = c(\widehat{\mathbf{U}}_2\backslash_B(\widehat{\mathbf{L}}_2\backslash_F(\mathbf{q_2} - \widehat{\mathbf{H}}_{21}(\widehat{\mathbf{U}}_1^{-1}(\widehat{\mathbf{L}}_1^{-1}\mathbf{q_1})))))$
4: compute $\mathbf{r_1} = \widehat{\mathbf{U}}_1^{-1}(\widehat{\mathbf{L}}_1^{-1}(c\mathbf{q_1} - \widehat{\mathbf{H}}_{12}\mathbf{r_2}))$
5: create $\mathbf{r}$ by concatenating $\mathbf{r_1}$ and $\mathbf{r_2}$
6: **return r**

Hence, $\Delta\mathbf{S}$ is expressed by the following equation:

$$\Delta\mathbf{S} = \Delta\mathbf{H}_{22} - \mathbf{H}_{21}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\Delta\mathbf{H}_{12})). \tag{11}$$

After computing $\Delta\mathbf{S}$, we simply update $\mathbf{S}$ to $\widehat{\mathbf{S}}$ by adding $\Delta\mathbf{S}$ as described in Algorithm 6.

4.1.5. Computing $\widehat{\mathbf{L}}_2$ *and* $\widehat{\mathbf{U}}_2$. In this step, we compute $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$, which are the updated matrices of $\mathbf{L_2}$ and $\mathbf{U_2}$, respectively. At the previous step, we already obtain $\widehat{\mathbf{S}}$, which is the updated matrix of $\mathbf{S}$. Hence, we decompose $\widehat{\mathbf{S}}$ into $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ using LU decomposition.

Note that we do not invert $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ because inverting a matrix is time consuming. To improve the performance of the update phase, we keep only $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$. We will explain how BEARD uses $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ to compute RWR scores in Section 4.2.

4.2. Query Phase

In the query phase, BEARD computes the RWR score vector $\hat{\mathbf{r}}$ with respect to a given seed node s with the updated matrices $\widehat{\mathbf{D}}$, $\widehat{\mathbf{H}}$, $\widehat{\mathbf{L}}_1^{-1}$, $\widehat{\mathbf{U}}_1^{-1}$, $\widehat{\mathbf{S}}$, $\widehat{\mathbf{L}}_2$, and $\widehat{\mathbf{U}}_2$. Algorithm 7 describes the query phase of BEARD.

In BEARD, we need to modify the query phase of BEARS because the query phase demands $\widehat{\mathbf{L}}_2^{-1}$ and $\widehat{\mathbf{U}}_2^{-1}$, but we have $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$. To exploit $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$, BEARD uses forward and backward substitution algorithms [Boyd and Vandenberghe 2009], which solve $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a lower or upper triangular matrix. Using the substitution algorithms, BEARD computes RWR scores with $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ under the same time complexity of BEARS. We will prove the time complexity of the query phase of BEARD in Section 4.3.1.

To express the substitution algorithms, we define operators $\backslash_F$ and $\backslash_B$, which indicate the forward substitution and the backward substitution, respectively.

Definition 4.4. (*Backslash Operator for a Triangular Matrix*). Suppose that $\mathbf{L}$ is an $n \times n$ invertible lower triangular matrix, $\mathbf{U}$ is an $n \times n$ invertible upper triangular matrix, and $\mathbf{b}$ and $\mathbf{x}$ are $n \times 1$ vectors. Then, a backslash operator for $\mathbf{L}$, $\backslash_F$, solves a linear system $\mathbf{Lx} = \mathbf{b}$ using the *forward substitution algorithm* and is denoted as follows:

$$\mathbf{Lx} = \mathbf{b} \Leftrightarrow \mathbf{x} = \mathbf{L}\backslash_F\mathbf{b}.$$

Another backslash operator for $\mathbf{U}$, $\backslash_B$, solves a linear system $\mathbf{Ux} = \mathbf{b}$ using the *backward substitution algorithm* and is denoted as follows:

$$\mathbf{Ux} = \mathbf{b} \Leftrightarrow \mathbf{x} = \mathbf{U}\backslash_B\mathbf{b}.$$

The vector $\mathbf{q}$ in Algorithm 7 is the length-n starting vector as mentioned in Section 3.2. With the updated matrices and the backslash operators, the exact RWR score vector $\mathbf{r}$ is computed by the following equation:

$$\mathbf{r} = \begin{bmatrix} \mathbf{r_1} \\ \mathbf{r_2} \end{bmatrix} = \begin{bmatrix} \widehat{\mathbf{U}}_1^{-1}(\widehat{\mathbf{L}}_1^{-1}(c\mathbf{q_1} - \widehat{\mathbf{H}}_{12}\mathbf{r_2})) \\ c(\widehat{\mathbf{U}}_2 \backslash_B (\widehat{\mathbf{L}}_2 \backslash_F (\mathbf{q_2} - \widehat{\mathbf{H}}_{21}(\widehat{\mathbf{U}}_1^{-1}(\widehat{\mathbf{L}}_1^{-1}\mathbf{q_1}))))) \end{bmatrix}. \tag{12}$$

THEOREM 4.5 (THE CORRECTNESS OF BEARD). *The* $\mathbf{r}$ *in Equation (12) is equal to the* $\mathbf{r}$ *in* $\widehat{\mathbf{H}}\mathbf{r} = c\mathbf{q}$.

PROOF. The $\mathbf{r}$ is represented with the updated matrices $\widehat{\mathbf{H}}$, $\widehat{\mathbf{L}}_1^{-1}$, $\widehat{\mathbf{U}}_1^{-1}$, $\widehat{\mathbf{L}}_2$, and $\widehat{\mathbf{U}}_2$ using Equation (6) as follows:

$$\mathbf{r} = \begin{bmatrix} \mathbf{r_1} \\ \mathbf{r_2} \end{bmatrix} = \begin{bmatrix} \widehat{\mathbf{U}}_1^{-1}(\widehat{\mathbf{L}}_1^{-1}(c\mathbf{q_1} - \widehat{\mathbf{H}}_{12}\mathbf{r_2})) \\ c(\widehat{\mathbf{U}}_2^{-1}(\widehat{\mathbf{L}}_2^{-1}(\mathbf{q_2} - \widehat{\mathbf{H}}_{21}(\widehat{\mathbf{U}}_1^{-1}(\widehat{\mathbf{L}}_1^{-1}\mathbf{q_1}))))) \end{bmatrix}. \tag{13}$$

Then, $\mathbf{r}$ in Equation (13) satisfies $\widehat{\mathbf{H}}\mathbf{r} = c\mathbf{q}$ by Theorem 3.4. However, we have $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$, instead of $\widehat{\mathbf{L}}_2^{-1}$ and $\widehat{\mathbf{U}}_2^{-1}$, as the results of the update phase. $\hat{\mathbf{r}}_2$ can be rewritten with $\widehat{\mathbf{L}}_2$, $\widehat{\mathbf{U}}_2$, and backslash operators as follows:

$$\begin{aligned} \widehat{\mathbf{L}}_2\widehat{\mathbf{U}}_2\mathbf{r_2} &= c(\mathbf{q_2} - \widehat{\mathbf{H}}_{21}(\widehat{\mathbf{U}}_1^{-1}(\widehat{\mathbf{L}}_1^{-1}\mathbf{q_1}))) \\ \mathbf{r_2} &= c(\widehat{\mathbf{U}}_2 \backslash_B (\widehat{\mathbf{L}}_2 \backslash_F (\mathbf{q_2} - \widehat{\mathbf{H}}_{21}(\widehat{\mathbf{U}}_1^{-1}(\widehat{\mathbf{L}}_1^{-1}\mathbf{q_1}))))). \quad \square \end{aligned}$$

BEARD answers a query using Equation (12), which exploits forward and backward substitutions; on the other hand, BEARS answers a query using Equation (6), which uses the inverses of $\mathbf{L_2}$ and $\mathbf{U_2}$.

One might wonder, why does BEARD use forward/backward substitutions without computing the inverses as in BEARS? The main reason is that the update phase of BEARD is performed *repeatedly*, whereas the preprocessing phase of BEARS is performed *only once*. Note that once the inverses are computed, computing $\mathbf{r}_2$ based on sparse matrix vector multiplication with $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$ requires fewer computations than those of forward/backward substitutions. This is because sparse matrix vector multiplication only considers nonzero entries in the sparse vector and the corresponding columns of the sparse matrix, whereas the substitution algorithms need to scan all nonzero entries in the matrix to compute the solution. Once computed, the inverses are used to speed up the query phase in BEARS. However, computing the inverses in each update phase of BEARD would be too time consuming. Thus, BEARD chooses not to compute the inverses. Furthermore, the query times of BEARD and BEARS do not differ drastically, as we will see in Section 5.8.

There is one more reason that BEARS computes the inverse: BEARS's approximate version BEARS-APPROX needs to investigate $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$ to drop near-zero entries of them as mentioned in Section 3.1.4.

4.3. Complexity Analysis

In this section, we analyze the time and the space complexities of BEARD.

4.3.1. Time Complexity. Here, we prove the time complexities for update and query phases of BEARD. The time complexity of each step of BEARD is summarized in Table IV.

LEMMA 4.6. *It takes* $O(m)$ *to update* $\mathbf{D}$ *and* $\mathbf{H}$.

PROOF. We update $\mathbf{H}$ into $\widehat{\mathbf{H}}$ by adding $\Delta\mathbf{H}$ to $\mathbf{H}$ (line 8 of Algorithm 4). For given two sparse matrices $\mathbf{A}$ and $\mathbf{B}$, it takes $O(|\mathbf{A}| + |\mathbf{B}|)$ to add them since we need to scan

Table IV. Time Complexity of Each Step of BEARD

Line	Task	Case	Time Complexity
	Update Phase (Algorithm 3)		
1	Updating $\mathbf{D}$ and $\mathbf{H}$ (Algorithm 4)	All	$O(m)$
2	Updating $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ (Algorithm 5)	Case 1 Case 2	$O(n_{1i_u}^3)$ $O((\sum_{i=1}^{b} n_{1i}^2) + min(n_1n_2, m) + n_{1i_{uv}}^3)$
3	Updating $\mathbf{S}$ (Algorithm 6)	Case 1 Case 2 Case 3	$O(n_{1i_u}^2 n_2 + n_{1i_u} n_2^2)$ $O(n_{1i_{uv}}^2 n_2 + n_{1i_{uv}} n_2^2)$ $O((\sum_{i=1}^{b} n_{1i}^2) + min(n_1n_2, m))$
4	Computing $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$	All	$O(n_2^3)$
Total	Case 1: $O(m + n_{1i_u}^3 + n_{1i_u}^2 n_2 + n_{1i_u} n_2^2 + n_2^3)$ Case 2: $O(m + (\sum_{i=1}^{b} n_{1i}^2) + min(n_1n_2, m) + n_{1i_{uv}}^3 + n_{1i_{uv}}^2 n_2 + n_{1i_{uv}} n_2^2 + n_2^3)$ Case 3: $O(m + (\sum_{i=1}^{b} n_{1i}^2) + min(n_1n_2, m) + n_2^3)$		
	Query Phase (Algorithm 7)		
3	Computing $\mathbf{r_2}$		$O((\sum_{i=1}^{b} n_{1i}^2) + n_2^2 + \min(n_1n_2, m))$
4	Computing $\mathbf{r_1}$		$O((\sum_{i=1}^{b} n_{1i}^2) + \min(n_1n_2, m))$
Total	$O((\sum_{i=1}^{b} n_{1i}^2) + n_2^2 + \min(n_1n_2, m))$		

all nonzero elements in $\mathbf{A}$ and $\mathbf{B}$. Since $\mathbf{H}$ has m nonzeros, $\Delta\mathbf{H}$ has $|N(u)|$ nonzeros, and $m > |N(u)|$, it takes $O(m)$ to add $\mathbf{H}$ and $\Delta\mathbf{H}$. □

LEMMA 4.7. *It takes* $O(n_{1i_u}^3)$ *for Case 1 and* $O((\sum_{i=1}^{b} n_{1i}^2) + min(n_1n_2, m) + n_{1i_{uv}}^3)$ *for Case 2 to update* $\mathbf{L_1^{-1}}$ *and* $\mathbf{U_1^{-1}}$.

PROOF. For Case 1, the algorithm updates $(\mathbf{L}_1^{i_u})^{-1}$ and $(\mathbf{U}_1^{i_u})^{-1}$. It takes $O(n_{1i_u}^3)$ to LU decompose $\widehat{\mathbf{H}}_{11}^{i_u}$ and invert the LU factors where n_{1i_u} is the number of nodes belonging to the block for $\widehat{\mathbf{H}}_{11}^{i_u}$. For Case 2, the matrices $\mathbf{D}$, $\widehat{\mathbf{H}}_{11}$, $\widehat{\mathbf{H}}_{12}$, $\widehat{\mathbf{H}}_{21}$, $\widehat{\mathbf{L}}_1^{-1}$, and $\widehat{\mathbf{U}}_1^{-1}$ need to be reordered. The number of edges in $\widehat{\mathbf{H}}_{11}$, $\widehat{\mathbf{L}}_1^{-1}$, and $\widehat{\mathbf{U}}_1^{-1}$ is $O(\sum_{i=1}^{b} n_{1i}^2)$; the number of edges in $\widehat{\mathbf{H}}_{12}$ and $\widehat{\mathbf{H}}_{21}$ is $O(min(n_1n_2, m))$. Then, the algorithm decomposes $\widehat{\mathbf{H}}_{11}^{i_{uv}}$ and inverts the LU factors in $O(n_{1i_{uv}}^3)$. □

LEMMA 4.8 (TIME COMPLEXITY FOR COMPUTING Γ). *Suppose that* $\mathbf{A}_{11}$ *is an* $n_1 \times n_1$ *block-diagonal matrix of* $\mathbf{A}$, n_{1i} *is the number of nodes in the ith block of* $\mathbf{A}_{11}$, $\mathbf{A}_{12}^i$ *is an* $n_{1i} \times n_2$ *matrix, and* $\mathbf{A}_{21}^i$ *is an* $n_2 \times n_{1i}$ *matrix. Then, it takes* $O(n_{1i}^2 n_2 + n_{1i} n_2^2)$ *to compute* $\Gamma(\mathbf{A}, i)$.

PROOF. By Definition 4.3 and Lemma 3.5, if the number of nonzeros of $(\mathbf{L_1}^i)^{-1}$ or $(\mathbf{U_1}^i)^{-1}$ is $O(n_{1i}^2)$, it takes $O(n_{1i}^2 n_2)$ to compute $\mathbf{R}_1 = (\mathbf{L_1}^i)^{-1}\mathbf{A}_{12}^i$, $O(n_{1i}^2 n_2)$ to compute $\mathbf{R}_2 = (\mathbf{U_1}^i)^{-1}\mathbf{R}_1$, and $O(n_{1i} n_2^2)$ to compute $\Gamma(\mathbf{A}, i) = -\mathbf{A}_{21}^i \mathbf{R}_2$. □

LEMMA 4.9. *For Case 1, it takes* $O(n_{1i_u}^2 n_2 + n_{1i_u} n_2^2)$ *to update* $\mathbf{S}$. *For Case 2, it takes* $O(n_{1i_{uv}}^2 n_2 + n_{1i_{uv}} n_2^2)$.

PROOF. By Lemma 4.8, it takes $O(n_{1i_u}^2 n_2 + n_{1i_u} n_2^2)$ to compute $\Gamma(\mathbf{H}, i_u)$ and $\Gamma(\widehat{\mathbf{H}}, i_u)$ for Case 1. And it takes $O(n_{1i_{uv}}^2 n_2 + n_{1i_{uv}} n_2^2)$ to compute $\Gamma(\mathbf{H}, i_u)$, $\Gamma(\mathbf{H}, i_v)$, and $\Gamma(\widehat{\mathbf{H}}, i_{uv})$ for Case 2. Note that $n_{1i_{uv}} > n_{1i_u}$ and $n_{1i_{uv}} > n_{1i_v}$. □

Table V. Maximum Number of Nonzero Entries in the Updated Matrices

Matrix	Max Nonzeros
$\widehat{\mathbf{D}}$	$O(n)$
$\widehat{\mathbf{H}}$	$O(m)$
$\widehat{\mathbf{H}}_{12}$ & $\widehat{\mathbf{H}}_{21}$	$O(\min(n_1 n_2, m))$
$\widehat{\mathbf{L}}_1^{-1}$ & $\widehat{\mathbf{U}}_1^{-1}$	$O(\sum_{i=1}^{b} n_{1i}^2)$
$\widehat{\mathbf{L}}_2$ & $\widehat{\mathbf{U}}_2$	$O(n_2^2)$

LEMMA 4.10. *For Case 3, it takes* $O((\sum_{i=1}^{b} n_{1i}^2) + min(n_1 n_2, m))$ *to update* $\mathbf{S}$.

PROOF. Since $\Delta\mathbf{H}_{12}$ contains only one column, it takes $O(\sum_{i=1}^{b} n_{1i}^2)$ to compute $\mathbf{R}_1 = \mathbf{L_1^{-1}}\Delta\mathbf{H}_{12}$ and $\mathbf{R}_2 = \mathbf{U_1^{-1}}\mathbf{R}_1$. In addition, since $\mathbf{R}_2$ contains only one column, it takes $O(min(n_1 n_2, m))$ to compute $\mathbf{H}_{21}\mathbf{R}_2$. Hence, the time complexity to compute $\Delta\mathbf{S}$ for Case 3 is $O((\sum_{i=1}^{b} n_{1i}^2) + min(n_1 n_2, m))$. □

THEOREM 4.11. *The update phase in* BEARD *takes* $O(m + n_{1i_u}^3 + n_{1i_u}^2 n_2 + n_{1i_u} n_2^2 + n_2^3)$ *for Case 1,* $O(m + (\sum_{i=1}^{b} n_{1i}^2) + min(n_1 n_2, m) + n_{1i_{uv}}^3 + n_{1i_{uv}}^2 n_2 + n_{1i_{uv}} n_2^2 + n_2^3)$ *for Case 2, and* $O(m + (\sum_{i=1}^{b} n_{1i}^2) + min(n_1 n_2, m) + n_2^3)$ *for Case 3.*

PROOF. See Lemmas 4.6, 4.7, 4.9, and 4.10, and Table IV. □

In the query phase of BEARD, we use forward and backward substitution algorithms to exploit $\widehat{\mathbf{L}}_1$ and $\widehat{\mathbf{U}}_1$. We first analyze the time complexity of the substitution algorithms.

LEMMA 4.12 (SPARSE FORWARD OR BACKWARD SUBSTITUTION). *Suppose that* $\mathbf{A}$ *is an* $n \times n$ *matrix,* $\mathbf{x}$ *and* $\mathbf{b}$ *are* $n \times 1$ *vectors, and* $\mathbf{A}$ *has* $|\mathbf{A}|$ *nonzero entries. If* $\mathbf{A}$ *is lower triangular, then sparse forward substitution takes* $O(|\mathbf{A}|)$. *In the case that* $\mathbf{A}$ *is upper triangular, sparse backward substitution also takes* $O(|\mathbf{A}|)$.

PROOF. The forward substitution algorithm scans nonzero entries on the ith row of $\mathbf{A}$ to compute the solution of the ith entry of $\mathbf{x}$. From the 1st entry to nth entry of $\mathbf{x}$, it scans all nonzero entries of $\mathbf{A}$. Similarly, the backward substitution algorithm scans all nonzero entries of $\mathbf{A}$ from the nth entry to the 1st entry of $\mathbf{x}$. □

THEOREM 4.13. *The query phase in* BEARD *takes* $O((\sum_{i=1}^{b} n_{1i}^2) + n_2^2 + \min(n_1 n_2, m))$.

PROOF. Apply Lemmas 3.5 and 4.12, and the results in Table V, to the query phase. □

As presented in Theorems 3.6 and 4.11, for all cases, the update phase of BEARD requires fewer computations than the preprocessing phase of BEARS. In the query phase, the time complexity of BEARD is the same as that of BEARS according to Theorems 3.7 and 4.13.

4.3.2. Space Complexity.

THEOREM 4.14. BEARD *requires* $O(n + m + \sum_{i=1}^{b} n_{1i}^2 + n_2^2)$ *memory space for updated matrices.*

PROOF. See Table V. □

5. EXPERIMENTS

In this section, we present experimental results. First, to evaluate the effectiveness of our exact method BearS-Exact, we design and conduct experiments that answer the following questions:

—*Q1. Preprocessing cost of BearS (Section 5.2):* How much memory space do BearS-Exact and its competitors require for their precomputed results? How long does this preprocessing phase take?
—*Q2. Query cost of BearS (Section 5.3):* How quickly does BearS-Exact answer an RWR query compared to other methods?
—*Q3. Effects of network structure for BearS (Section 5.4):* How does network structure affect the preprocessing time, query time, and space requirements of BearS-Exact?

Second, for our approximate method BearS-Approx, our experiments answer the following questions:

—*Q4. Effects of drop tolerance for BearS (Section 5.4):* How does drop tolerance ξ affect the accuracy, query time, and space requirements of BearS-Approx?
—*Q5. Comparison with approximate methods and BearS (Section 5.6):* Does BearS-Approx provide a better trade-off between accuracy, time, and space compared to its competitors?

Finally, to measure the performance of our dynamic method BearD, we design experiments to answer the following questions:

—*Q6. Update cost of BearD (Section 5.7):* Compared to preprocessing the changed graph, how long does the update phase take after one edge modification?
—*Q7. Query cost of BearD (Section 5.8):* How quickly does BearD answer an RWR query after one edge modification?

5.1. Experimental Settings

Machine. The experiments shown later for Figures 11(a), 11(c), 17(a), and 17(c) are conducted on a PC with a four-core Intel i5-4590 3.3GHz CPU and 16GB memory. All other experiments are performed in a similar PC (Intel i5-4570 3.2GHz CPU, and all other conditions are the same).

Data. The graph data used in our experiments are summarized in Table VI. A brief description of each real-world dataset is presented in Appendix A.4. For synthetic graphs, we use R-MAT [Chakrabarti et al. 2004b] with different p_{ul}, the probability that an edge falls into the upper-left partition. The probabilities for the other partitions are set to $(1 - p_{ul})/3$, respectively.

Implementation. We compare our methods to the iterative method, RPPR [Gleich and Polito 2006], BRPPR [Gleich and Polito 2006], inversion, LU decomposition [Fujiwara et al. 2012a], QR decomposition [Fujiwara et al. 2012b], B_LIN [Tong et al. 2008], and NB_LIN [Tong et al. 2008], all of which are explained in Section 2.2. All methods including Bear are implemented using MATLAB, which provides a state-of-the-art linear algebra package. In particular, our implementation of NB_LIN and that of RPPR optimize their open-sourced implementations[1,2] in terms of preprocessing speed and query speed, respectively.

Parameters. We set the restart probability c to 0.05 as in the previous work [Tong et al. 2008].[3] We set k of SlashBurn to $0.001n$ (see Appendix A.1 for the meaning of

[1]http://www.cs.cmu.edu/~htong/pdfs/FastRWR_20080319.tgz.
[2]http://www.mathworks.co.kr/matlabcentral/fileexchange/11613-pagerank.
[3]In this work, c denotes (1 – restart probability).

Table VI. Summary of Real-World and Synthetic Datasets

Dataset	n	m	n_2	$\sum_{i=1}^{b} n_{1i}^2$	$\lvert\mathbf{H}\rvert$	$\lvert\mathbf{H_{12}}\rvert + \lvert\mathbf{H_{21}}\rvert$	$\lvert\mathbf{L_1^{-1}}\rvert + \lvert\mathbf{U_1^{-1}}\rvert$	$\lvert\mathbf{L_2^{-1}}\rvert + \lvert\mathbf{U_2^{-1}}\rvert$
Routing†	22,963	48,436	572	678,097	119,835	72,168	86,972	271,248
Co-author†	31,163	120,029	4,464	1,364,443	271,221	118,012	202,482	18,526,862
Trust*	131,828	841,372	10,087	3,493,773	972,627	317,874	318,461	81,039,727
Email*	265,214	420,045	1,590	566,435	684,170	358,458	554,681	1,149,462
Web-Stan*	281,903	2,312,497	16,017	420,658,754	2,594,400	1,423,993	26,191,040	55,450,105
Web-Notre*	325,729	1,497,134	12,350	77,441,937	1,795,408	611,408	5,912,673	12,105,579
Web-BS*	685,230	7,600,595	50,005	717,727,201	8,285,825	4,725,657	25,990,336	547,936,698
Talk*	2,394,385	5,021,410	19,152	3,272,265	7,415,795	3,996,546	4,841,878	246,235,838
Citation*	3,774,768	16,518,948	1,143,522	71,206,030	20,293,715	9,738,225	6,709,469	408,178,662
R-MAT (0.5)*	99,982	500,000	17,721	1,513,855	599,982	184,812	204,585	260,247,890
R-MAT (0.6)*	99,956	500,000	11,088	1,258,518	599,956	145,281	205,837	104,153,333
R-MAT (0.7)*	99,707	500,000	7,029	705,042	599,707	116,382	202,922	43,250,097
R-MAT (0.8)*	99,267	500,000	4,653	313,848	599,267	104,415	199,576	19,300,458
R-MAT (0.9)*	98,438	500,000	3,038	244,204	598,438	107,770	196,873	8,633,841

†Undirected graph.
*Directed graph.
Note: $\lvert\mathbf{M}\rvert$ denotes the number of nonzero entries in the matrix $\mathbf{M}$. A brief description of each real-world dataset is presented in Appendix A.4.

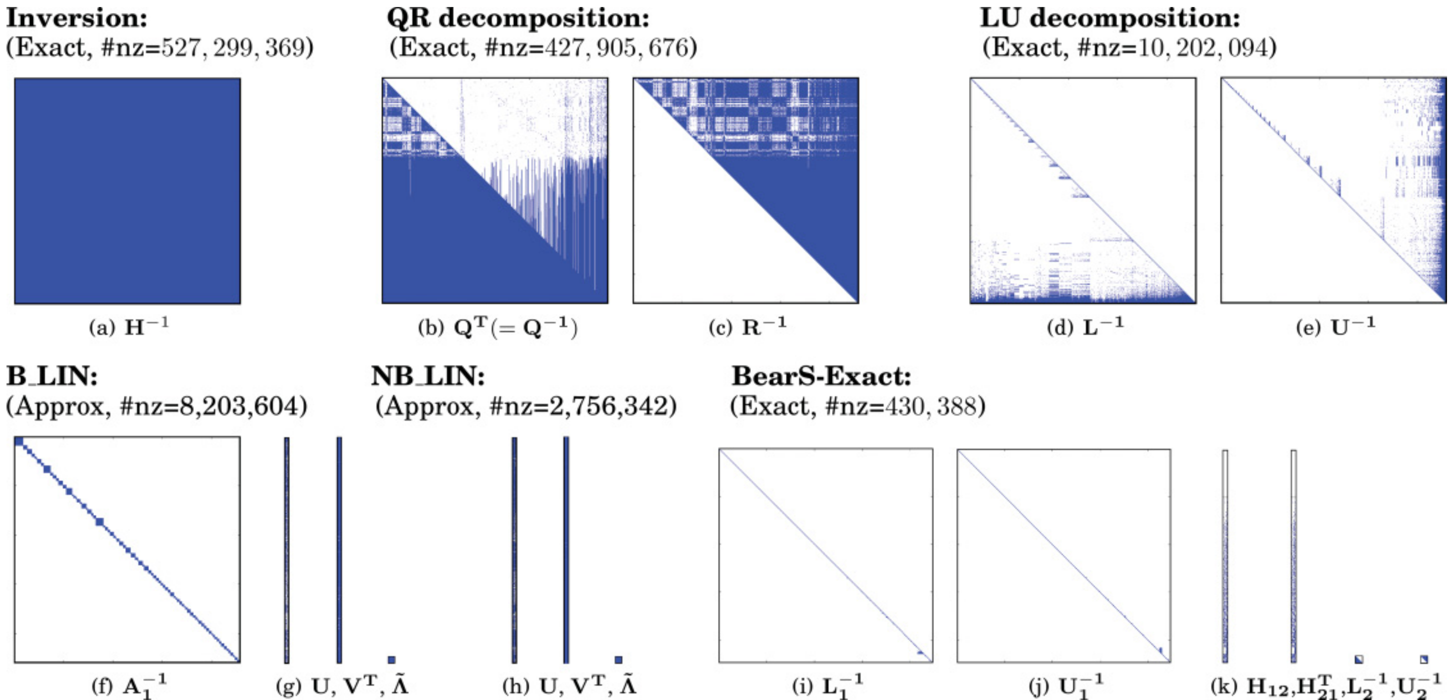
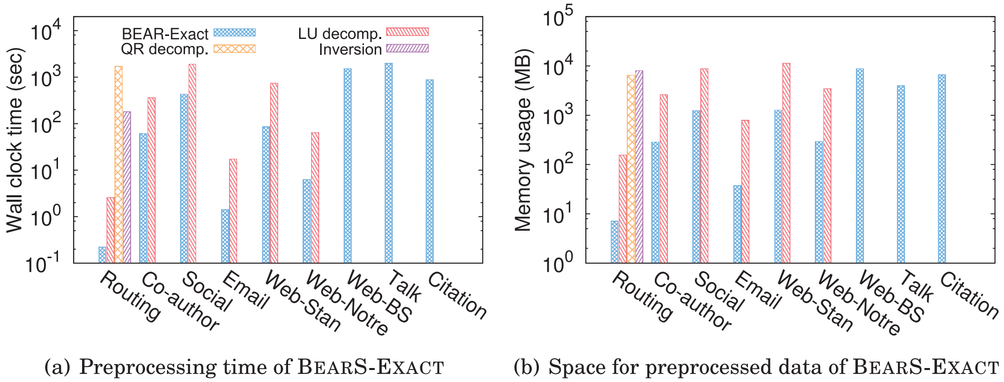


Fig. 4. Sparsity patterns of the matrices resulted from different preprocessing methods on the Routing dataset (see Section 5.1 for details of the dataset). Exact: exact method, Approx: approximate method, #nz: number of nonzero elements in the precomputed matrices. For B_LIN and NB_LIN, rank t is set to 500 and drop tolerance ξ is set to 0. Precomputed matrices are loaded into memory to speed up the query phase; for this reason, the number of nonzero entries in them determines the memory usage of each method and thus its scalability. Our exact method BEARS-EXACT produces the smallest number of nonzero entries than all other methods, including the approximate methods (1200× than Inversion and 6× than the second best method). Our approximate method BEARS-APPROX further decreases the number of nonzero entries, depending on drop tolerance ξ.



(a) Preprocessing time of BEARS-EXACT (b) Space for preprocessed data of BEARS-EXACT

Fig. 5. Preprocessing cost of BEARS-EXACT. In (a) and (b), bars are omitted if and only if the corresponding experiments run out of memory. (a) In the preprocessing phase, BEARS-EXACT is the fastest and the most scalable among all preprocessing methods. (b) BEARS-EXACT requires the least amount of space for preprocessed data on all datasets. Due to its space efficiency, only BEARS-EXACT successfully scales to the largest Citation dataset (with 3.8M nodes) without running out of memory.

k), which achieves a good trade-off between running time and reordering quality. The convergence threshold ϵ of the iterative method is set to 10^{-8}, which gives accurate results. For B_LIN and NB_LIN, we use the heuristic decomposition method proposed in their work, which is much faster with little difference in accuracy compared to SVD in our experiments. The number of partitions in B_LIN, the rank in B_LIN and NB_LIN, and the convergence threshold of RPPR and BRPPR are tuned for each dataset, which are summarized later in Table VIII in Appendix A.5.

5.2. Preprocessing Cost of BEARS

We compare the preprocessing cost of BEARS-EXACT with that of other exact methods. Figure 5(a) and (b) present the preprocessing time and space requirements of the methods except the iterative method, which does not require preprocessing. Only BEARS-EXACT successfully preprocesses all datasets, whereas others fail due to their high memory requirements.

Preprocessing time is measured in wall clock time and includes time for SlashBurn (in BEARS-EXACT) and community detection (in LU decomposition). BEARS-EXACT requires the least amount of time, which is less than an hour, for all datasets, as seen in Figure 5(a). Especially in the graphs with distinct hub-and-spoke structure, BEARS-EXACT is up to 12× faster than the second best one.

To compare space-efficiency, we measure the amount of memory required for precomputed matrices of each method. The precomputed matrices are saved in the compressed sparse column format [Press 2007], which requires memory space proportional to the number of nonzero entries. As seen in Figure 5(b), BEARS-EXACT requires up to 22× less memory space than its competitors in all the datasets, which results in the superior scalability of BEARS-EXACT compared to the competitors, which can be verified in Figure 4 as well.

We analyze the trade-off of BEARS-EXACT between space for preprocessed data and preprocessing time. Figure 7(a) shows the results on the datasets. In the figure, BEARS-EXACT is more closely located in the lower-left area than other methods. This result indicates that the preprocessing phase of BEARS-EXACT provides a better trade-off between space and time.

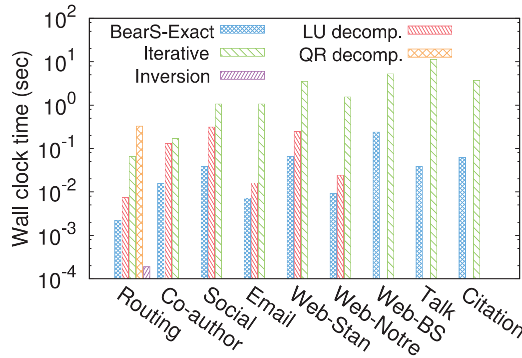


Fig. 6. Query cost of BearS-Exact. Bars are omitted if the corresponding methods run out of memory for preprocessing. Note that BearS-Exact is the fastest except for the smallest Routing dataset where the inversion method runs faster. However, the inversion method fails to process other datasets due to its limited scalability.

5.3. Query Cost of BearS

We compare BearS-Exact with other exact methods in terms of query time, which means time taken to compute $\mathbf{r}$ for a given seed node. Although time taken for a single query is small compared to preprocessing time, reducing query time is important because many applications require RWR scores for different query nodes (e.g., all nodes) or require the real-time computation of RWR scores for a query node.

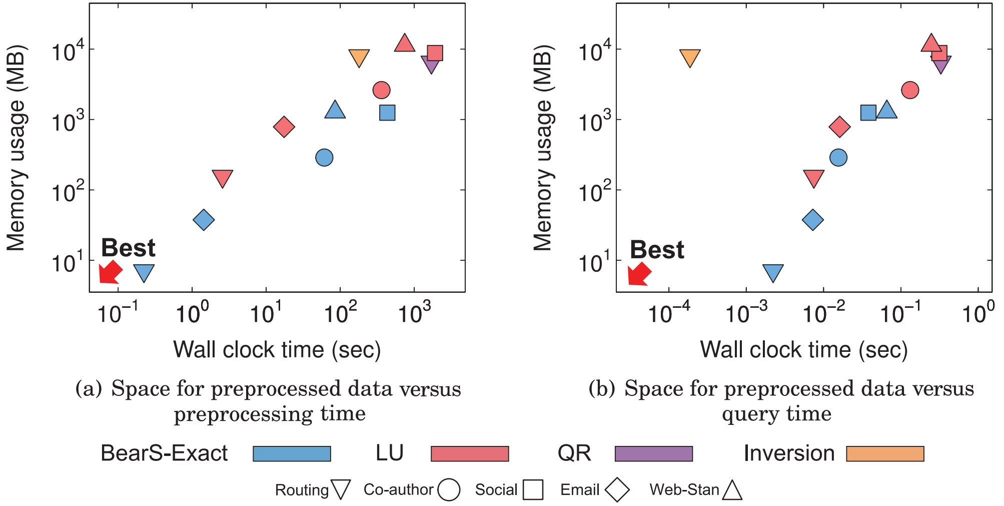
Figure 6 shows the result where the y-axis represents the average query time for 1,000 random seed nodes. Only BearS-Exact and the iterative method run successfully on all graphs; the others fail on large graphs due to their high space requirements. BearS-Exact outperforms its competitors in all datasets except the smallest one, which is the only dataset to which the inversion method can scale. BearS-Exact is up to 8× faster than LU decomposition, the second best one, and in the Talk dataset, it is almost 300× faster than the iterative method, which is the only competitor. Although BearS-Exact requires a preprocessing step that is not needed by the iterative method, for real-world applications where RWR scores for many query nodes are required, BearS-Exact outperforms the iterative method in terms of total running time.

Figure 7(b) shows the trade-off of BearS-Exact between memory space and query time. The points corresponding to BearS-Exact are closer to the lower-left area than other competitors, indicating that the query phase of BearS-Exact provides a good trade-off between time and space. On the Routing dataset, the inversion method computes queries faster than BearS-Exact, but the inversion method requires much more memory space than BearS-Exact and fails to run on other larger datasets.

Furthermore, BearS-Exact also provides the best performance in PPR, where the number of seeds is greater than one. In Appendix B.1, Figure 14 provides the comparison of the query time of BearS-Exact with that of others in PPR; in the same appendix, Figure 15 provides the query time of BearS-Exact with different numbers of seeds.

5.4. Effects of Network Structure for BearS

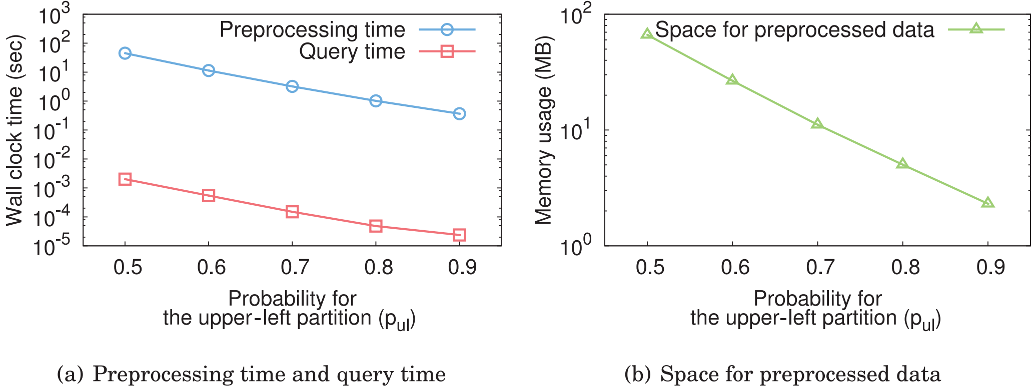
The complexity analysis in Section 3.3 indicates that the performance of BearS-Exact depends on the structure of a given graph. Specifically, the analysis implies that BearS-Exact is fast and space efficient on a graph with strong hub-and-spoke structure where the graph is divided into small pieces by removing a small number of hubs. In this experiment, we empirically support this claim using synthetic graphs with similar sizes but different structures.



(a) Space for preprocessed data versus preprocessing time

(b) Space for preprocessed data versus query time

Fig. 7. Trade-off between time and space of BEARS-EXACT. The colors distinguish the methods, and the shapes distinguish the datasets. The iterative method does not appear in the figures because it does not require space for preprocessed data. The Web-BS, Talk, and Citation datasets are excluded from the figures because only BEARS-EXACT preprocesses the datasets. In both figures, the lower-left region indicates better performance. BEARS-EXACT is located more closely to those regions than the competitors in most of the datasets.



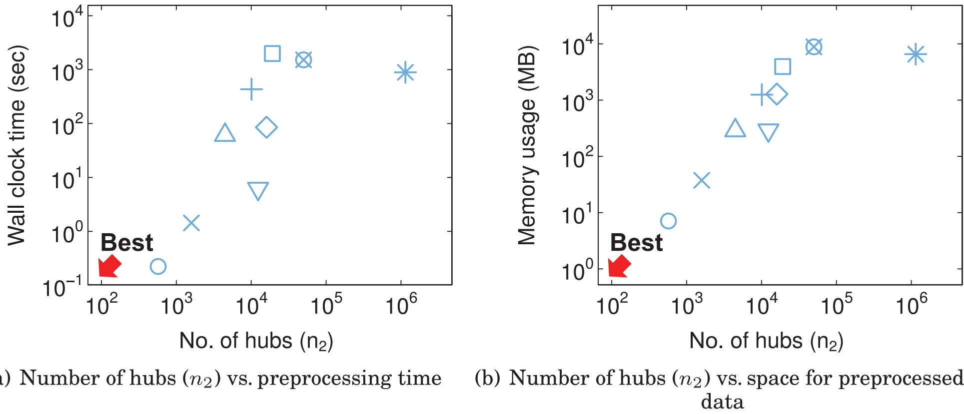
(a) Preprocessing time and query time

(b) Space for preprocessed data

Fig. 8. Effects of network structure on synthetic graphs. BEARS-EXACT becomes fast and space efficient on a graph with a distinct hub-and-spoke structure (a graph with high p_{ul}).

Table VI summarizes four synthetic graphs generated using R-MAT [Chakrabarti et al. 2004b] with different p_{ul}, the probability that an edge falls into the upper-left partition. As p_{ul} increases, the hub-and-spoke structure becomes stronger, as seen from the number of hubs (n_2) and the size of partitions ($\sum_{i=1}^{b} n_{1i}^2$) of each graph. Figure 8(a) and (b) show the performance of BEARS-EXACT on these graphs. Preprocessing time, query time, and space required for preprocessed data decline rapidly with regard to p_{ul}, which is coherent with what the complexity analysis implies.

The experimental results on real-world graphs also support the complexity analysis. Figure 9(a) and (b) show preprocessing time and memory usage of BEARS-EXACT with regard to n_2 on real-world graphs. As seen in the figure, datasets with large n_2 such as Web-BS, Talk, and Citation demand more preprocessing time and memory space than those with small n_2 such as Routing, Co-author, and Web-Notre.



(a) Number of hubs (n_2) vs. preprocessing time (b) Number of hubs (n_2) vs. space for preprocessed data

Fig. 9. Effects of network structure on real-world graphs. BEARS-EXACT becomes fast and space efficient as the number of hubs n_2 decreases.

5.5. Effects of Drop Tolerance for BEARS

In this experiment, we measure the effects of different drop tolerance values on the query time, space requirements, and accuracy of BEARS-APPROX. We change the drop tolerance, ξ, from 0 to $n^{-1/4}$ and measure the accuracy using cosine similarity[4] and L2-norm of error[5] between $\mathbf{r}$ computed by BEARS-EXACT and $\hat{\mathbf{r}}$ computed by BEARS-APPROX with the given drop tolerance. Figure 10 summarizes the results. We observe that both the space required for preprocessed data and the query time of BEARS-APPROX significantly decrease compared to those of BEARS-EXACT, and the accuracy remains high.

5.6. Comparison of Approximate Methods and BEARS

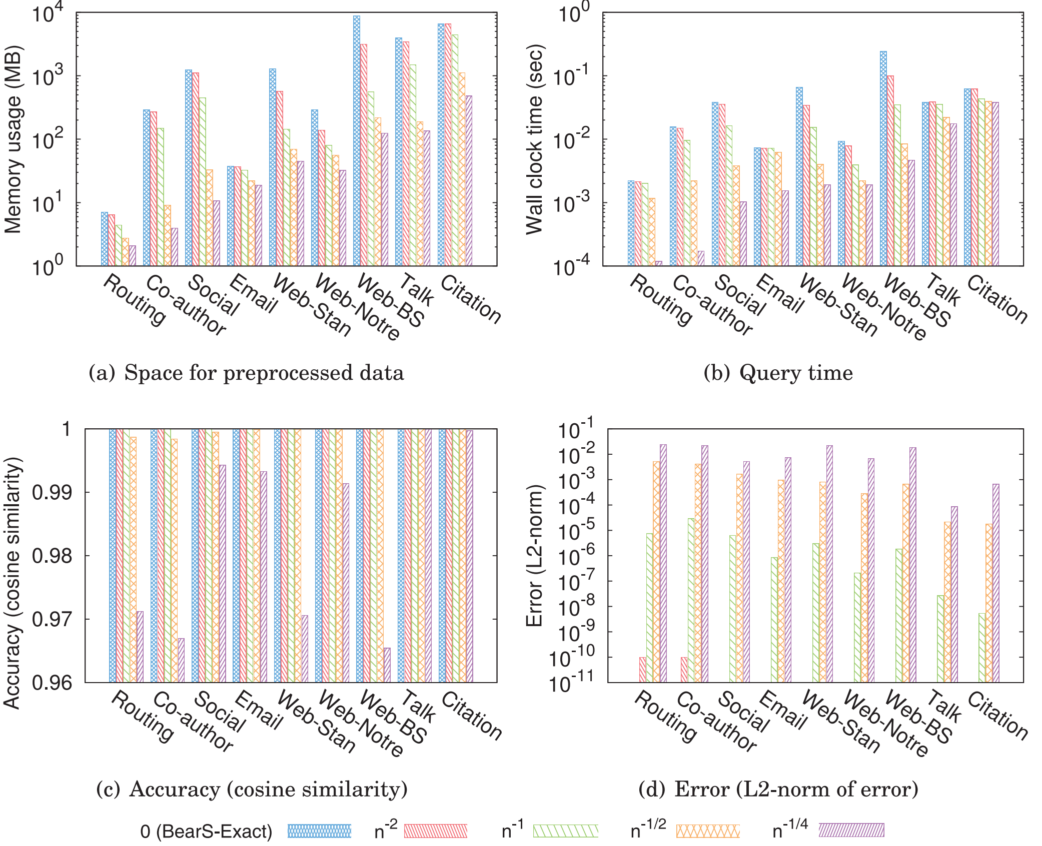
We conduct performance comparison among BEARS-APPROX and other state-of-the-art approximate methods. Dropping near-zero entries of precomputed matrices is commonly applied to BEARS-APPROX, B_LIN, and NB_LIN, providing a trade-off between accuracy, query time, and storage cost. We analyze this trade-off by changing drop tolerance from 0 to $n^{-1/4}$. Likewise, we analyze the trade-off between accuracy and time provided by RPPR, BRPPR, and Push by changing the threshold, θ_b, from 10^{-4} to 0.5. RPPR, BRPPR, and Push do not require space for preprocessed data. For Push, we conduct experiments on undirected graphs because Push only works on undirected graphs. Accuracy is measured using cosine similarity[4] and L2-norm of error[5] as in Section 5.4.

Figure 11 summarizes the result on two datasets. According to the figure, BEARS-APPROX provides a better trade-off between accuracy, time, and space than other competitors. The results on other datasets are given later in Figure 17 of Appendix B.3. The preprocessing times of the approximate methods are also compared in Figure 16 of Appendix B.2.

5.7. Update Cost of BEARD

We conduct experiments to measure the performance of the update phase of BEARD. Figure 12(a) compares the preprocessing time of BEARS and the update time of BEARD.

[4] $(\mathbf{r}\cdot\hat{\mathbf{r}})/(||\mathbf{r}||\,||\hat{\mathbf{r}}||)$, ranging from -1 (dissimilar) to 1 (similar).
[5] $||\hat{\mathbf{r}}-\mathbf{r}||$, ranging from 0 (no error) to $||\hat{\mathbf{r}}|| + ||\mathbf{r}||$ (max error bound).



(a) Space for preprocessed data (b) Query time

(c) Accuracy (cosine similarity) (d) Error (L2-norm of error)

Fig. 10. Effects of drop tolerance on the performance of BearS-Approx. Drop tolerance changes from 0 to $n^{-1/4}$. As drop tolerance increases, space requirements and query time are significantly improved, and accuracy, measured by cosine similarity and L2-norm of error, remains high.

Note that updating the preprocessed matrices using BearD takes less time than re-preprocessing the modified graph using BearS.

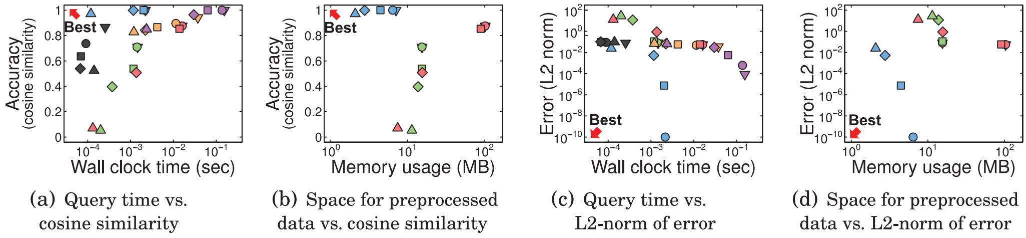
We first preprocess a given graph using BearS and then randomly generate 30 edges for each case. Then, for each case, we measure the average update time per one edge using BearD. In addition, we measure the total average update time for all cases. The preprocessing and update times are measured in wall clock time.

As shown in Figure 12(a), updating the preprocessed matrices using BearD is faster than re-preprocessing the changed graph using BearS. In particular, BearD updates the given matrices about 29× faster for the Citation dataset. Overall, the update times of all cases are similar for most datasets. This is because LU decomposition of $\widehat{\mathbf{S}}$ occupies the major part of the update time, which requires $O(n_2^3)$. This result indicates that other factors of the time complexity in Theorem 4.11 are dominated by $O(n_2^3)$.

5.8. Query Cost of BearD

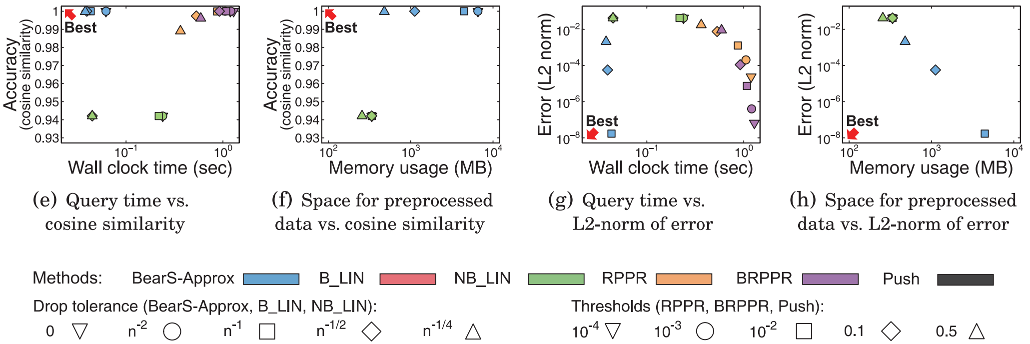
We compare BearD with BearS in terms of query time. First, we preprocess a graph and incrementally update the preprocessed matrices case by case as mentioned in Section 5.7. For each case, we measure the average query time for 1,000 random seeds and then calculate the total average query time for all cases.

Routing:



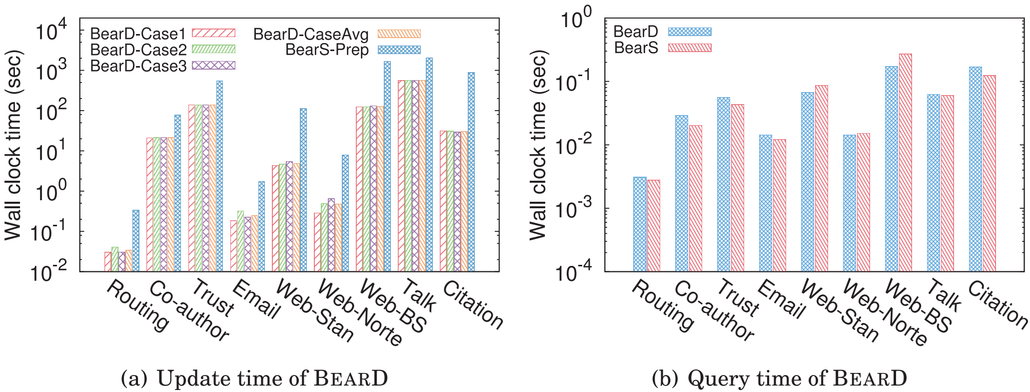
(a) Query time vs. cosine similarity (b) Space for preprocessed data vs. cosine similarity (c) Query time vs. L2-norm of error (d) Space for preprocessed data vs. L2-norm of error

Citation:



(e) Query time vs. cosine similarity (f) Space for preprocessed data vs. cosine similarity (g) Query time vs. L2-norm of error (h) Space for preprocessed data vs. L2-norm of error

Fig. 11. Trade-off between time, space, and accuracy provided by approximate methods. Parts (a) through (d) show the results on the Routing dataset, which is an undirected graph, and parts (e) through (h) show the results on the Citation dataset, which is a directed graph. The colors distinguish the methods, and the shapes distinguish the drop tolerance (for BEARS-APPROX, B_LIN, and NB_LIN) or the threshold (for RPPR, BRPPR, and Push). RPPR, BRPPR, and Push do not appear in the figures in the second and fourth columns because they do not require space for preprocessed data. In addition, Push only appears in figures on undirected graphs because the method only works on undirected graphs. In the left four figures, the upper-left region indicates better performance, whereas in the right four figures, the lower-left region indicates better performance. Notice that BEARS-APPROX provides (1) the best trade-off between accuracy and query speed and (2) the best trade-off between accuracy and space requirements among preprocessing methods.



(a) Update time of BEARD (b) Query time of BEARD

Fig. 12. Performance of BEARD. (a) Preprocessing time of BEARS and the updating time of BEARD for each case. Updating the preprocessed matrices using BEARD is faster than preprocessing the changed graph in all datasets. (b) BEARD computes a given query almost as fast as BEARS, even though BEARS is slightly faster than BEARD.

Table VII. Comparison Between the Number of Nonzero Entries of $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ and That of $\widehat{\mathbf{L}}_2^{-1}$ and $\widehat{\mathbf{U}}_2^{-1}$ After 30 Updates Using BEARD

Dataset	$\lvert\widehat{\mathbf{L}}_2\rvert + \lvert\widehat{\mathbf{U}}_2\rvert$	$\lvert\widehat{\mathbf{L}}_2^{-1}\rvert + \lvert\widehat{\mathbf{U}}_2^{-1}\rvert$	$\frac{\lvert\widehat{\mathbf{L}}_2^{-1}\rvert + \lvert\widehat{\mathbf{U}}_2^{-1}\rvert}{\lvert\widehat{\mathbf{L}}_2\rvert + \lvert\widehat{\mathbf{U}}_2\rvert}$
Routing	146, 209	273, 937	1.87
Co-author	11, 336, 514	18, 531, 856	1.63
Trust	40, 946, 464	81, 039, 732	1.98
Email	487, 670	1, 149, 684	2.36
Web-Stan	6, 001, 812	55, 814, 600	9.30
Web-Notre	932, 674	12, 977, 961	13.91
Web-BS	75, 187, 228	643, 052, 606	8.55
Talk	103, 517, 667	246, 235, 838	2.38
Citation	101, 497, 101	409, 041, 447	4.03

Note: $\lvert\mathbf{M}\rvert$ denotes the number of nonzero entries in the matrix $\mathbf{M}$.

As seen in Figure 12(b), BEARD is comparable with BEARS in terms of query speed in all datasets. For most datasets, BEARD is slightly slower than BEARS because of forward and backward substitution algorithms. BEARD is up to 1.4× slower than BEARS for the reasons described in Section 4.2. However, BEARD is faster than BEARS for other datasets, such as Web-Stan, Web-Notre, and Web-BS. In this case, BEARD is up to 1.5× faster than BEARS. This is because for those datasets, the number of nonzero entries in $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ is much smaller than that of $\widehat{\mathbf{L}}_2^{-1}$ and $\widehat{\mathbf{U}}_2^{-1}$. According to Table VII, the number of nonzero entries of $\widehat{\mathbf{L}}_2^{-1}$ and $\widehat{\mathbf{U}}_2^{-1}$ is at least 8.55× and at most 13.91× greater than that of $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$. Note that BEARD computes RWR scores with $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$, and BEARS computes the scores with $\widehat{\mathbf{L}}_2^{-1}$ and $\widehat{\mathbf{U}}_2^{-1}$.

This result justifies that BEARD uses the substitution algorithms with $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$ because the difference between the query time of BEARD and that of BEARS is small. In addition, if $\widehat{\mathbf{L}}_2^{-1}$ and $\widehat{\mathbf{U}}_2^{-1}$ are much denser than $\widehat{\mathbf{L}}_2$ and $\widehat{\mathbf{U}}_2$, BEARD computes queries with fewer computations than BEARS.

6. RELATED WORK

In this section, we review related work, which can be categorized into four parts: (1) relevance measures and applications, (2) approximate methods for RWR, (3) top-k search for RWR, and (4) preprocessing methods for RWR.

Relevance measures and applications. There are various relevance (proximity) measures based on random walk, such as penalized hitting probability [Zhang et al. 2012], EI [Bogdanov and Singh 2013], discounted hitting time [Sarkar and Moore 2010], truncated hitting time [Sarkar and Moore 2007], RWR [Pan et al. 2004], effective conductance [Doyle and Snell 1984], ObjectRank [Balmin et al. 2004], SimRank [Jeh and Widom 2002], and PPR [Page et al. 1999]. Among these measures, RWR has received much interest and has been applied to many applications, including community detection [Andersen et al. 2006; Gleich and Seshadhri 2012; Whang et al. 2013; Zhu et al. 2013], ranking [Tong et al. 2008], link prediction [Backstrom and Leskovec 2011], graph matching [Tong et al. 2007; Kang et al. 2012], knowledge discovery [Kasneci et al. 2009], anomaly detection [Sun et al. 2005], content-based image retrieval [He et al. 2004], and cross-modal correlation discovery [Pan et al. 2004]. Andersen et al. [2006] proposed a local community detection algorithm that utilizes RWR to find a cut with small conductance near a seed node. This algorithm was used to explore the properties of communities in large graphs because it is fast and returns tight communities [Leskovec et al. 2009]. Considerable improvements of the algorithm have been made

regarding seed finding [Gleich and Seshadhri 2012; Whang et al. 2013] and inner connectivity [Zhu et al. 2013]. Backstrom and Leskovec [2011] proposed a link prediction algorithm called *supervised random walk*, which is a variant of RWR. In the algorithm, transition probabilities are determined as a function of the attributes of nodes and edges, and the function is adjusted through supervised learning. Tong et al. [2007] proposed G-Ray, which finds best-effort subgraph matches for a given query graph in a large and attributed graph. They formulated a goodness function using RWR to estimate the proximity between a node and a subgraph. Kasneci et al. [2009] exploited RWR as a measure of node-based informativeness to extract an informative subgraph for given query nodes. Sun et al. [2005] utilized RWR for neighborhood formulation and abnormal node detection in bipartite graphs. He et al. [2004] employed RWR to rank retrieved images in their manifold ranking algorithm. Pan et al. [2004] proposed a method adopting RWR to compute the correlations between a query image node and caption nodes.

Approximate methods for RWR. The iterative method, which comes from the definition of RWR, is not fast enough in real-world applications where RWR scores for different query nodes need to be computed. To overcome this obstacle, approximate approaches have been proposed. Sun et al. [2005] observed that the relevance scores for a seed node are highly skewed, and many real-world graphs have a block-wise structure. Based on these observations, they performed random walks only on the partition containing the seed node and assigned the proximities of zero to the other nodes outside the partition. Instead of using a precomputed partition, Gleich and Polito [2006] proposed methods that adaptively determine the part of a graph used for RWR computation in the query phase, as explained in Section 2.2. Andersen et al. [2006] also proposed an approximate method based on local information. Tong et al. [2008] proposed approximate approaches called *B_LIN* and *NB_LIN*. They achieved higher accuracy than previous methods by applying a low-rank approximation to cross-partition edges instead of ignoring them. Lofgren et al. [2014] proposed a Monte Carlo–based method, FAST-PPR, which estimates PPR of a single pair between a start node and a target node. Their method first finds two sets: a target set and a frontier set. The target set contains nodes whose RWR score to the target node is greater than a threshold, and the frontier set contains nodes within one step from the boundary of the target set. The method estimates the single-pair PPR score by doing random walks from the start node with testing if the walks hit the frontier set.

Top-k search for RWR. Several recent works focus on finding the k most relevant nodes of a seed node instead of calculating the RWR scores of every node. Gupta et al. [2008] proposed the basic push algorithm (BPA), which finds top-k nodes for PPR in an efficient way. BPA precomputes relevance score vectors with respect to hub nodes and uses them to obtain the upper bounds of PPR scores. Fujiwara et al. [2012a] proposed K-dash, which computes the RWR scores of top-k nodes exactly. It computes the RWR scores efficiently by exploiting precomputed sparse matrices and pruning unnecessary computations while searching for the top-k nodes. Wu et al. [2014] showed that many random walk–based measures have the no-local-minimum (or no-local-maximum) property, which means that each node, except for a given query node, has at least one neighboring node having lower (or higher) proximity. Based on this property, they developed a unified local search method called *fast local search* (FLoS), which exactly finds top-k relevant nodes in terms of measures satisfying the no-local-optimum property. Furthermore, Wu et al. showed that FLoS can be applied to RWR, which does not have the no-local-optimum property, by utilizing its relationship with other measures. Bahmani et al. [2010] proposed a Monte Carlo method that estimates top-k nodes for PPR based on random walk segments. However, top-k RWR computation

is insufficient for many data mining applications [Andersen et al. 2006; Backstrom and Leskovec 2011; Gleich and Seshadhri 2012; He et al. 2004; Sun et al. 2005; Tong et al. 2007; Whang et al. 2013; Zhu et al. 2013] that demand the RWR scores of all nodes.

Preprocessing methods for RWR. As seen from Section 2.2, the computational cost of RWR can be significantly reduced by precomputing $\mathbf{H}^{-1}$. However, matrix inversion does not scale up. In other words, for a large graph, it often results in a dense matrix that cannot fit to memory. For this reason, alternative methods have been proposed. NB_LIN, proposed by Tong et al. [2007], decomposes the adjacency matrix using low-rank approximation in the preprocessing phase and approximates $\mathbf{H}^{-1}$ from these decomposed matrices in the query phase using the Sherman-Morrison Lemma [Piegorsch and Casella 1990]. Its variant, B_LIN, uses this technique only for cross-partition edges. These methods require less space but do not guarantee accuracy. Fujiwara et al. inverted the results of LU decomposition [Fujiwara et al. 2012a] or QR decomposition [Fujiwara et al. 2012b] of $\mathbf{H}$ after carefully reordering nodes. Their methods produce sparser matrices that can be used in place of $\mathbf{H}^{-1}$ in the query phase but still have limited scalability. Furthermore, these preprocessing methods are improper when graphs evolve because the methods should repeatedly perform the expensive preprocessing phase.

In our previous work [Shin et al. 2015], we developed a preprocessing method for static graphs. The method exactly computes RWR scores by exploiting the hub-spoke structure of real-world graphs and block elimination. Moreover, we proposed an approximate method that drops near-zero entries of the preprocessed matrices. According to our extensive experiments, our methods outperformed other preprocessing and approximate approaches in terms of accuracy, time, and space. Even though our previous methods have shown better scalability and fast query speed, they are inappropriate for dynamic graphs because the time-consuming preprocessing phase should be repetitively executed whenever the graphs change. Contrary to the previous methods, our proposed BearD quickly updates the preprocessed matrices when edges are inserted or deleted, and it immediately computes queries based on the update matrices. In addition, we showed that both BearS and BearD are applicable to FaBP with a wider convergence range, as described in Section 3.4.

In addition to the approaches described earlier, distributed computing is another promising approach. Andersen et al. [2012] proposed a distributed platform to solve linear systems including RWR. To reduce communication cost, they partitioned a graph into overlapping clusters and assigned each cluster to a distinct processor. Additionally, computing random walk–based proximities on dynamic graphs is an interesting problem. Bahmani et al. [2010] proposed a Monte Carlo–based method for incremental computation of PageRank. Their method first performs random walks at each node, stores the random walk segments, and estimates approximate PageRank scores based on the segments. When edges are added or removed, the method updates the random walk segments relevant to the added or removed edges.

7. CONCLUSION

In this article, we propose Bear, a novel algorithm for fast, scalable, and accurate RWR computation on large graphs. Bear comprises BearS and BearD; BearS is a preprocessing method for static graphs, and BearD is an incremental update method for dynamic graphs. We discuss the two versions of BearS: BearS-Exact and BearS-Approx. The former guarantees accuracy, whereas the latter is faster and more space efficient with little loss of accuracy. For dynamic graphs, BearD analyzes the changes of the given graph when an edge modification occurs and quickly updates the modified parts of the preprocessed matrices. We experimentally show that the preprocessing phase of

the exact method BEARS-EXACT takes up to 12× less time and requires up to 22× less memory space than that of other preprocessing methods guaranteeing accuracy, which makes BEARS-EXACT enjoy the superior scalability. BEARS-EXACT also outperforms the competitors in the query phase: it is up to 8× faster than other preprocessing methods, and in large graphs where other preprocessing methods run out of memory, it is almost 300× faster than its only competitor—the iterative method. The approximate method BEARS-APPROX consistently provides a better trade-off between accuracy, time, and storage cost compared to other approximate methods. The update method BEARD is designed for dynamic graphs that change over time: BEARD is up to 29× faster for processing dynamic graphs than BEARS. Future research directions include extendingBEAR to handle very large graphs on distributed systems.

A. APPENDIX

A.1. Details of SlashBurn

SlashBurn [Kang and Faloutsos 2011; Lim et al. 2014] is a node reordering method for a graph so that the nonzeros of the resulting adjacency matrix are concentrated. For the purpose, SlashBurn first removes k hub nodes (high-degree nodes) from the graph so that the graph is divided into the GCC and the remaining disconnected components. Then, SlashBurn reorders nodes such that the hub nodes get the highest node IDs, the nodes in the disconnected components get the lowest node IDs, and the nodes in the GCC get the IDs in the middle. The preceding procedure repeats on the GCC until the size of the GCC becomes smaller than k. When SlashBurn finishes, the adjacency matrix of the graph contains a large and sparse block-diagonal matrix in the upper-left area, as shown in Figure 2(b). Figure 13 illustrates this process when $k = 1$.

A.2. The Proof of Lemma 3.2

PROOF. Let $\mathbf{L}'$ and $\mathbf{U}'$ be the block-diagonal matrices that consist of $\mathbf{L_1}$ through $\mathbf{L_b}$ and $\mathbf{U_1}$ through $\mathbf{U_b}$, respectively. Then, $\mathbf{L} = \mathbf{L}'$ and $\mathbf{U} = \mathbf{U}'$ because (1) $\mathbf{L}'$ is a unit lower triangular matrix; (2) $\mathbf{U}'$ is an upper triangular matrix; (3) $\mathbf{L}'\mathbf{U}'$ is the block-diagonal matrix consisting of $\mathbf{L_1U_1}$ through $\mathbf{L_bU_b}$ that is equal to $\mathbf{A}$; and (4) $\mathbf{L}'$ and $\mathbf{U}'$ satisfying (1) through (3) are the unique LU decomposition of $\mathbf{A}$ [Banerjee and Roy 2014]. The multiplication of $\mathbf{L}$ and $\mathbf{L}^{-1}$ defined in Lemma 3.2 results in the block-diagonal matrix consisting of $\mathbf{L_1L_1^{-1}}$ through $\mathbf{L_bL_b^{-1}}$ that is an identity matrix. Likewise, the multiplication of $\mathbf{U}$ and $\mathbf{U}^{-1}$ defined in Lemma 3.2 results in an identity matrix. □

A.3. The Proof of Convergence of FaBP

LEMMA 8.1 (INVERTIBLE CONDITION OF FABP). *The matrix* $\mathbf{H} = [\mathbf{I} + a\mathbf{D} - c'\mathbf{A}]$ *is strictly diagonally dominant if* $\frac{1}{2(1-max_i D_{ii})} < h_h < 0$*, or* $0 < h_h < \frac{1}{2(max_i D_{ii}-1)}$ *where* $a = \frac{4h_h^2}{1-4h_h^2}$, $c' = \frac{2h_h}{1-4h_h^2}$, $-\frac{1}{2} < h_h < \frac{1}{2}$*, and* $h_h \neq 0$.

PROOF. If $\mathbf{H}$ is a strictly diagonally dominant matrix, then the matrix is nonsingular or invertible. Hence, if $\mathbf{H}$ is invertible, the following inequality must be satisfied for all nodes:

$$\begin{aligned} |H_{ii}| &> \sum_{j \neq i} |H_{ij}| \text{ for each node } i \\ 1 + |a|D_{ii} &> |c'|D_{ii}, \end{aligned} \tag{14}$$

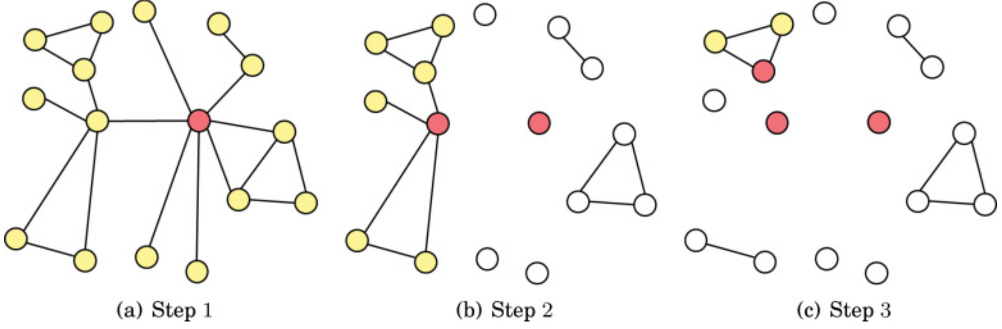


Fig. 13. Hub selection in SlashBurn when $k = 1$. Hub nodes are colored red, nodes in the GCC are colored yellow, and nodes in disconnected components are colored white.

where $|H_{ii}| = 1 + |a|D_{ii}$ and $\sum_{j \neq i} |H_{ij}| = |c'|D_{ii}$. If $-\frac{1}{2} < h_h < \frac{1}{2}$, then a is positive. For a variable c', if $0 < h_h < \frac{1}{2}$, then c' is positive. Otherwise, c' is negative if $-\frac{1}{2} < h_h < 0$.

Case 1. $-\frac{1}{2} < h_h < 0$. In this case, $a > 0$ and $c' < 0$. Then, Equation (14) is written as follows:

$$1 + aD_{ii} > -c'D_{ii}$$

$$1 + \frac{4h_h^2}{1 - 4h_h^2} D_{ii} + \frac{2h_h}{1 - 4h_h^2} D_{ii} > 0$$

$$h_h > \frac{1}{2(1 - D_{ii})}$$

$$h_h > \frac{1}{2(1 - max_i D_{ii})} \text{ for all nodes.}$$

Hence, if $\frac{1}{2(1-max_i D_{ii})} < h_h < 0$, then $\mathbf{H}$ is invertible.

Case 2. $0 < h_h < \frac{1}{2}$. In this case, $a > 0$ and $c' > 0$. Then, Equation (14) is written as follows:

$$1 + aD_{ii} > c'D_{ii}$$

$$1 + \frac{4h_h^2}{1 - 4h_h^2} D_{ii} - \frac{2h_h}{1 - 4h_h^2} D_{ii} > 0$$

$$h_h < \frac{1}{2(D_{ii} - 1)}$$

$$h_h < \frac{1}{2(max_i D_{ii} - 1)} \text{ for all nodes.}$$

Hence, if $0 < h_h < \frac{1}{2(max_i D_{ii}-1)}$, then $\mathbf{H}$ is invertible. Finally, if $\frac{1}{2(1-max_i D_{ii})} < h_h < 0$, or $0 < h_h < \frac{1}{2(max_i D_{ii}-1)}$, then $\mathbf{H}$ is invertible. □

A.4. Experimental Datasets

Next, we provide a short description of the real-world datasets used in Section 5.

—*Routing*[6]: The structure of the Internet at the level of autonomous systems. This data was reconstructed from BGP tables collected by the University of Oregon Route Views Project.[7]
—*Co-author*[8]: The co-authorship network of scientists who posted preprints on the Condensed Matter E-Print Archive.[9] The data include all preprints posted between January 1, 1995 and June 30, 2003.
—*Trust*[10]: A who-trust-whom online social network taken from Epinions.com,[11] a general consumer review site. Members of the site decide whether to trust each other, and this affects which reviews are shown to them.
—*Email*[12]: An email network taken from a large European research institution. The data include all incoming and outgoing emails of the research institution from October 2003 to May 2005.
—*Web-Notre*[13]: The hyperlink network of Web pages from the University of Notre Dame in 1999.
—*Web-Stan*[14]: The hyperlink network of Web pages from Stanford University in 2002.
—*Web-BS*[15]: The hyperlink network of Web pages from the University of California at Berkeley and Stanford University in 2002.
—*Talk*[16]: A who-talks-to-whom network taken from Wikipedia,[17] a free encyclopedia written collaboratively by volunteers around the world. The data include all users and discussions (talks) from the inception of Wikipedia to January 2008.
—*Citation*[18]: The citation network of utility patents granted in the United States between 1975 and 1999.

A.5. Parameter Settings for B_LIN, NB_LIN, RPPR, BRPPR, and Push

For each dataset, we determine the number of partitions ($\#p$) and the rank (t) of B_LIN among $\{100, 200, 500, 1000, 2000\}$ and $\{100, 200, 500, 1000\}$, respectively, so that their pair provides the best trade-off between query time, space for preprocessed data, and accuracy. Likewise, the rank (t) of NB_LIN is determined among $\{100, 200, 500, 1000\}$, and the convergence threshold (ϵ) of RPPR, BRPPR, and Push is determined among $\{10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$. The parameter values used for each dataset are summarized in Table VIII. B_LIN runs out of memory on the Talk dataset and the Citation dataset regardless of parameter values used. Push only works on undirected graphs.

B. ADDITIONAL EXPERIMENTS

B.1. Query Time in PPR

We measure the query time of exact methods when the number of seeds is greater than one, which corresponds to PPR. We change the number of seeds (nonzero entries in $\mathbf{q}$) from 1 to 1,000. As seen in Figure 14, BEARS-EXACT is the fastest method regardless of

[6]http://www-personal.umich.edu/~mejn/netdata/as-22july06.zip.
[7]http://www.routeviews.org/.
[8]http://www-personal.umich.edu/~mejn/netdata/cond-mat-2003.zip.
[9]http://arxiv.org/archive/cond-mat.
[10]http://snap.stanford.edu/data/soc-sign-epinions.html.
[11]http://www.epinions.com/.
[12]http://snap.stanford.edu/data/email-EuAll.html.
[13]http://snap.stanford.edu/data/web-NotreDame.html.
[14]http://snap.stanford.edu/data/web-Stanford.html.
[15]http://snap.stanford.edu/data/web-BerkStan.html.
[16]http://snap.stanford.edu/data/wiki-Talk.html.
[17]http://www.wikipedia.org/.
[18]http://snap.stanford.edu/data/cit-Patents.html.

Table VIII. Parameter Values of B_LIN, NB_LIN, RPPR, BRPPR, and Push Used for Each Dataset

	B_LIN		NB_LIN	RPPR	BRPPR	Push
Dataset	$\#p$	t	t	ϵ	ϵ	ϵ
Routing	200	200	100	10^{-4}	10^{-4}	10^{-4}
Co-author	200	500	1,000	10^{-4}	10^{-4}	10^{-3}
Trust	100	200	1,000	10^{-4}	10^{-5}	—
Email	1,000	100	200	10^{-3}	10^{-5}	—
Web-Stan	1,000	100	100	10^{-3}	10^{-4}	—
Web-Notre	500	100	200	10^{-4}	10^{-5}	—
Web-BS	2,000	100	100	10^{-3}	10^{-5}	—
Talk	—	—	200	10^{-3}	10^{-6}	—
Citation	—	—	100	10^{-4}	10^{-5}	—

Note: $\#p$ denotes the number of partitions, t denotes the rank, and ϵ denotes the convergence threshold.

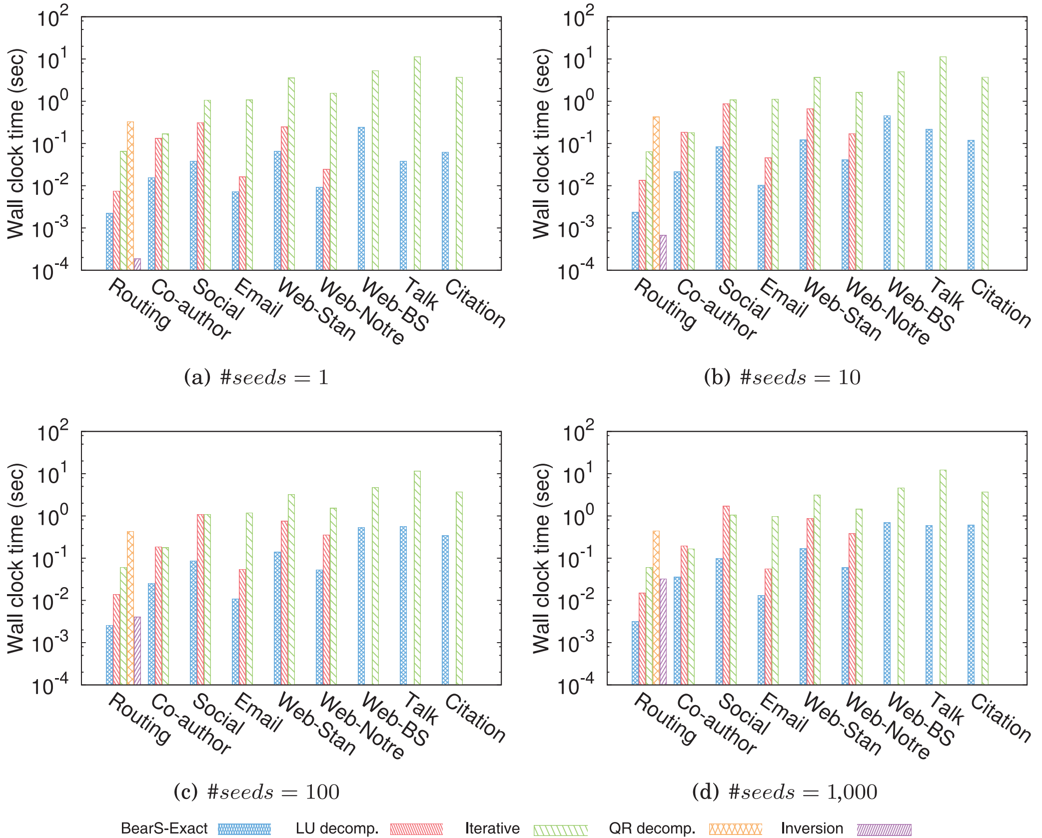


Fig. 14. Query time of exact methods with different numbers of seeds. If a method cannot scale to a dataset, the corresponding bar is omitted in the graphs. BearS-Exact is the fastest method regardless of the number of seeds in all datasets except the smallest Routing dataset.

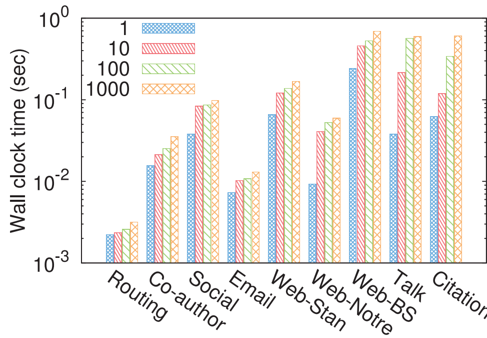


Fig. 15. Effect of the number of seeds on the query time of BEARS-EXACT. Query time increases as the number of seeds increases, but the rate of increase slows down.

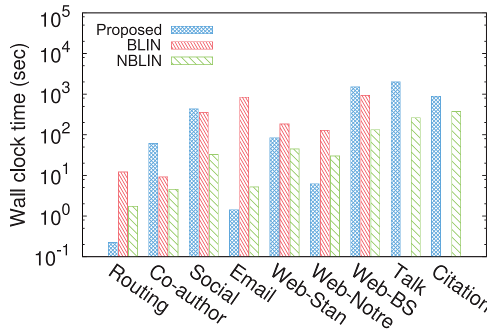


Fig. 16. Preprocessing time of approximate methods. The relative performance among the methods depends on the characteristics of data.

the number of seeds in all datasets except the smallest Routing dataset. The query time of the inverse method increases rapidly as the number of seeds increases. It increases by 210× on the Routing dataset, whereas the query time of BEARS-EXACT increases by 3×. Figure 15 summarizes the effect of the number of seeds on the query time of BEARS-EXACT. In most datasets, the effect of the number of seeds diminishes as the number of seeds increases. The query time increases by up to 16× depending on the number of seeds.

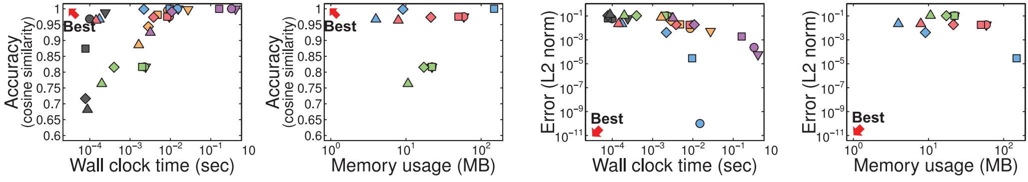
B.2. Preprocessing Time of Approximate Methods

Figure 16 presents the preprocessing time of approximate methods. Preprocessing time is measured in wall clock time and includes time for SlashBurn (in BEARS-APPROX) and community detection (in B_LIN). B_LIN cannot scale to the Talk dataset and the Citation dataset because it runs out of memory while inverting the block-diagonal matrices. The relative performance among the methods depends on the structure of graphs, as summarized in Table VI. BEARS-APPROX tends to be faster than the competitors on graphs with a small number of hubs (e.g., the Routing and Email datasets) and slower on those with a large number of hubs (e.g., the Web-BS and Citation datasets).

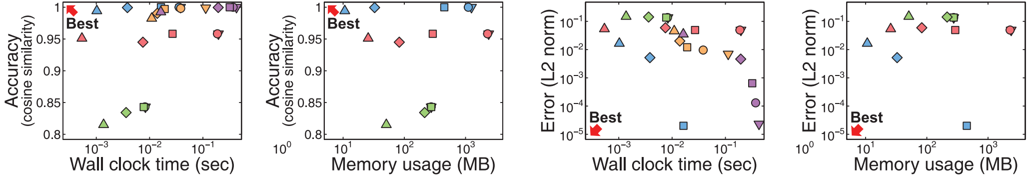
B.3. Comparison with Approximate Methods

Figure 17 shows the result of the experiments described in Section 5.6 on other datasets. In most of the datasets, BEARS-APPROX gives the best trade-off between accuracy and time. It also provides the best trade-off between accuracy and storage among all preprocessing methods.

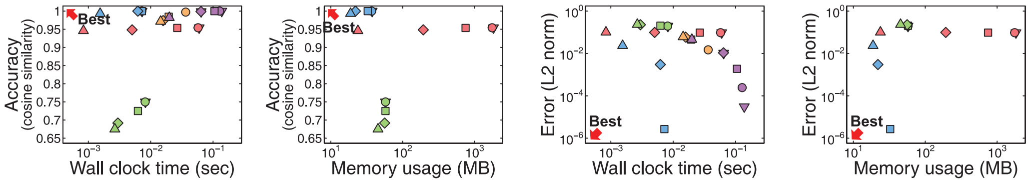
Co-author:



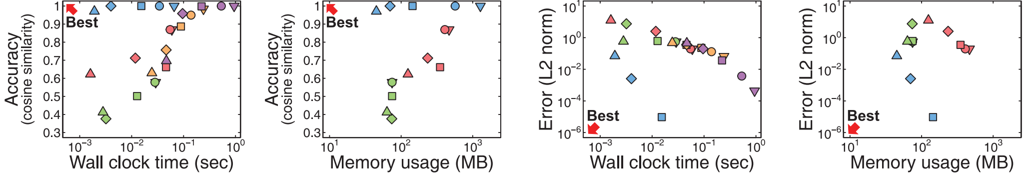
Trust:



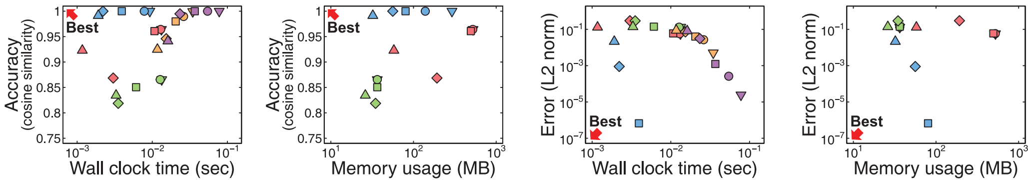
Email:



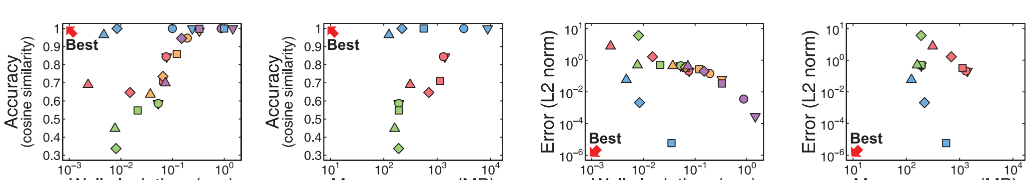
Web-Stan:



Web-Notre:



Web-BS:



(a) Query time vs. cosine similarity (b) Space for preprocessed data vs. cosine similarity (c) Query time vs. L2-norm of error (d) Space for preprocessed data vs. L2-norm of error

(Continued on next page)

Talk:

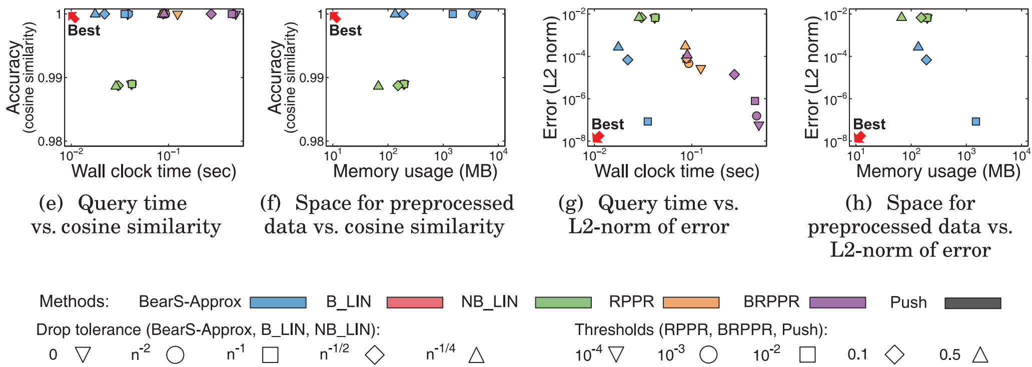


Fig. 17. Trade-off between time, space, and accuracy provided by approximate methods. Each row of figures shows the results on the corresponding dataset. The colors distinguish the methods, and the shapes distinguish the drop tolerance (for BEARS-APPROX, B_LIN, and NB_LIN) or the threshold (for RPPR, BRPPR, and Push). RPPR, BRPPR, and Push do not appear in the figures in the second and fourth columns because they do not require space for preprocessed data. In addition, Push only appears in figures for undirected graphs. In the figures in the left two columns, the upper-left region indicates better performance, whereas in the figures in the right two columns, the lower-left region indicates better performance. BEARS-APPROX is located more closely to those regions than the competitors in most of the datasets.

REFERENCES

Lada A. Adamic and Eytan Adar. 2003. Friends and neighbors on the Web. *Social Networks* 25, 3, 211–230.

Reka Albert, Hawoong Jeong, and Albert-Laszlo Barabasi. 2000. Error and attack tolerance of complex networks. *Nature* 406, 6794, 378–382.

Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local graph partitioning using PageRank vectors. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'06)*. 475–486.

Reid Andersen, David F. Gleich, and Vahab Mirrokni. 2012. Overlapping clusters for distributed computation. In *Proceedings of the International Conference on Web Search and Data Mining (WSDM'12)*. 273–282.

Ioannis Antonellis, Hector Garcia Molina, and Chi Chao Chang. 2008. Simrank++: Query rewriting through link analysis of the click graph. *Proceedings of the VLDB Endowment* 1, 1, 408–421.

Lars Backstrom and Jure Leskovec. 2011. Supervised random walks: Predicting and recommending links in social networks. In *Proceedings of the International Conference on Web Search and Data Mining (WSDM'11)*. 635–644.

Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast incremental and personalized PageRank. *Proceedings of the VLDB Endowment* 4, 3, 173–184.

Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. 2004. ObjectRank: Authority-based keyword search in databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'04)*. 564–575.

Sudipto Banerjee and Anindya Roy. 2014. *Linear Algebra and Matrix Analysis for Statistics*. CRC Press, Boca Raton, FL.

Petko Bogdanov and Ambuj Singh. 2013. Accurate and scalable nearest neighbors in large networks based on effective importance. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM'13)*. 1009–1018.

Stephen Boyd and Lieven Vandenberghe. 2009. *Convex Optimization*. Cambridge University Press.

Deepayan Chakrabarti, Spiros Papadimitriou, Dharmendra S. Modha, and Christos Faloutsos. 2004a. Fully automatic cross-associations. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'04)*. 79–88.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004b. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining (SDM'04)*. Vol. 4. 442–446.

Soumen Chakrabarti, Amit Pathak, and Manish Gupta. 2011. Index design and query processing for graph conductance search. *Proceedings of the VLDB Endowment* 20, 3, 445–470.

Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'09)*. 219–228.

Peter G. Doyle and J. Laurie Snell. 1984. *Random Walks and Electric Networks*. Mathematical Association of America.

Yasuhiro Fujiwara, Makoto Nakatsuji, Makoto Onizuka, and Masaru Kitsuregawa. 2012a. Fast and exact top-k search for random walk with restart. *Proceedings of the VLDB Endowment* 5, 5, 442–453.

Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka. 2012b. Efficient personalized PageRank with accuracy assurance. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'12)*. 15–23.

David Gleich and Marzia Polito. 2006. Approximating personalized PageRank with minimal use of Web graph data. *Internet Mathematics* 3, 3, 257–294.

David F. Gleich and C. Seshadhri. 2012. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'12)*. 597–605.

Manish Gupta, Amit Pathak, and Soumen Chakrabarti. 2008. Fast algorithms for top$k$ personalized PageRank queries. In *Proceedings of the International Conference on World Wide Web (WWW'08)*. 1225–1226.

F. Harary and G. Gupta. 1997. Dynamic graph models. *Mathematical and Computer Modelling* 25, 7, 79–87.

Jingrui He, Mingjing Li, Hong-Jiang Zhang, Hanghang Tong, and Changshui Zhang. 2004. Manifold-ranking based image retrieval. In *Proceedings of the Annual ACM International Conference on Multimedia (MULTIMEDIA'04)*. 9–16.

Glen Jeh and Jennifer Widom. 2002. SimRank: A measure of structural-context similarity. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'02)*. 538–543.

U. Kang and Christos Faloutsos. 2011. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. In *Proceedings of the International Conference on Data Mining (ICDM'11)*. 300–309.

U. Kang, H. Tong, and J. Sun. 2012. Fast random walk graph kernel. In *Proceedings of the SIAM International Conference on Data Mining (SDM'12)*. 828–838.

Gjergji Kasneci, Shady Elbassuoni, and Gerhard Weikum. 2009. Ming: Mining informative entity relationship subgraphs. In *Proceedings of the InternationalConference on Information and Knowledge Management (CIKM'09)*. 1653–1656.

Danai Koutra, Tai-You Ke, U. Kang, Duen Horng Chau, Hsing-Kuo Kenneth Pao, and Christos Faloutsos. 2011. Unifying guilt-by-association approaches: Theorems and fast algorithms. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD'11)*. 245–260.

Danai Koutra, Joshua T. Vogelstein, and Christos Faloutsos. 2013. DELTACON: A principled massive-graph similarity function. In *Proceedings of the 13th SIAM International Conference on Data Mining (SDM'13)*. 162–170.

Amy N. Langville and Carl D. Meyer. 2011. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ.

Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1, 29–123.

David Liben-Nowell and Jon Kleinberg. 2007. The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology* 58, 7, 1019–1031.

Y. Lim, U. Kang, and C. Faloutsos. 2014. SlashBurn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering* 26, 12, 3077–3089. DOI: http://dx.doi.org/10.1109/TKDE.2014.2320716

Zhenjiang Lin, Michael R. Lyu, and Irwin King. 2009. MatchSim: A novel neighbor-based similarity measure with maximum neighborhood matching. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM'09)*. 1613–1616.

P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. 2014. FAST-PPR: Scaling personalized PageRank estimation for large graphs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 1436–1445.

Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. 2002. On spectral clustering: Analysis and an algorithm. *Advances in Neural Information Processing Systems* 14, 2, 849–856.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report. Stanford University, Stanford, CA.

Jia-Yu Pan, Hyung-Jeong Yang, Christos Faloutsos, and Pinar Duygulu. 2004. Automatic multimedia cross-modal correlation discovery. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'04)*. 653–658.

Walter W. Piegorsch and George Casella. 1990. Inverting a sum of matrices. *SIAM Review* 32, 3, 470–470.

William H. Press. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press.

Purnamrita Sarkar and Andrew W. Moore. 2007. A tractable approach to finding closest truncated-commute-time neighbors in large graphs. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI'07)*. 335–343.

Purnamrita Sarkar and Andrew W. Moore. 2010. Fast nearest-neighbor search in disk-resident graphs. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'10)*. 513–522.

K. Shin, J. Jung, L. Sael, and U. Kang. 2015. BEAR: Block elimination approach for random walk with restart on large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*.

Jimeng Sun, Huiming Qu, Deepayan Chakrabarti, and Christos Faloutsos. 2005. Neighborhood formation and anomaly detection in bipartite graphs. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'05)*. 418–425.

Hanghang Tong and Christos Faloutsos. 2006. Center-piece subgraphs: Problem definition and fast solutions. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'06)*. 404–413.

Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. 2007. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'07)*. 737–746.

Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2008. Random walk with restart: Fast solutions and applications. *Knowledge and Information Systems* 14, 3, 327–346.

Joyce Jiyoung Whang, David F. Gleich, and Inderjit S. Dhillon. 2013. Overlapping community detection using seed set expansion. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM'13)*. 2099–2108.

Yubao Wu, Ruoming Jin, and Xiang Zhang. 2014. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. 1139–1150.

Chao Zhang, Lidan Shou, Ke Chen, Gang Chen, and Yijun Bei. 2012. Evaluating geo-social influence in location-based social networks. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM'12)*. 1442–1451.

Zeyuan A. Zhu, Silvio Lattanzi, and Vahab Mirrokni. 2013. A local algorithm for finding well-connected clusters. In *Proceedings of the International Conference on Machine Learning (ICML'13)*. 396–404.

Received May 2015; revised December 2015; accepted January 2016