Fast Nearest Neighbor Search on Large Time-Evolving Graphs

Leman Akoglu[1], Rohit Khandekar[2], Vibhore Kumar[3], Srinivasan Parthasarathy[3], Deepak Rajan[4], and Kun-Lung Wu[3]

[1] Stony Brook University `leman@cs.stonybrook.edu`
[2] Knight Capital Group `rkhandekar@gmail.com`
[3] IBM T. J. Watson Research {`vibhorek, spartha, klwu`}`@us.ibm.com`
[4] Lawrence Livermore National Labs `rdeepak@gmail.com`

Abstract. Finding the k nearest neighbors (k-NNs) of a given vertex in a graph has many applications such as link prediction, keyword search, and image tagging. An established measure of vertex-proximity in graphs is the Personalized Page Rank (PPR) score based on random walk with restarts. Since PPR scores have long-range correlations, computing them accurately and efficiently is challenging when the graph is too large to fit in main memory, especially when it also changes over time. In this work, we propose an efficient algorithm to answer PPR-based k-NN queries in large time-evolving graphs. Our key approach is to use a divide-and-conquer framework and efficiently compute answers in a distributed fashion. We represent a given graph as a collection of dense vertex-clusters with their inter connections. Each vertex-cluster maintains certain information related to internal random walks and updates this information as the graph changes. At query time, we combine this information from a small set of relevant clusters and compute PPR scores efficiently. We validate the effectiveness of our method on large real-world graphs from diverse domains. To the best of our knowledge, this is one of the few works that simultaneously addresses answering k-NN queries in possibly disk-resident *and* time-evolving graphs.

Keywords: vertex proximity · personalized pagerank · time-evolving graphs · disk-resident graphs · distributed pagerank · dynamic updates

1 Introduction

Quantifying the proximity, relevance, or similarity between vertices, and more generally finding the k nearest neighbors (k-NNs) of a given vertex in a large, time-evolving graph is a fundamental building block for many applications. Personalized PageRank (PPR) has proved to be a very effective proximity measure for the link prediction and recommendation problems in such applications. Thanks to its effectiveness, there exist many algorithms in the literature that are designed to compute the PPR scores of a given vertex in a graph efficiently [4, 10–13, 25]. These works, however, cannot handle graphs that are larger than a certain size, that is, they are not optimally designed for very large disk-resident graphs. Moreover, they cannot work with graphs that dynamically change over time. Other previous works deal with computing PPR queries on either disk-resident *static* graphs [21, 3] or special families of time-evolving graphs [26].

In this work we propose ClusterRank, an algorithm for efficient computation of PPR queries for *both* disk-resident *and* time-evolving general graphs. Our main motivation is to build a unified framework that will enable us to tackle both of these two challenges. Our key idea is to take a divide-and-conquer approach; simply put, we split the graph into relatively small vertex-clusters and decompose the overall problem into simulating intra-cluster and inter-cluster random walks. This decomposition enables us to handle disk-resident graphs since the work is carefully split across distributed compute nodes. What is more, thanks to this modular design of our approach, our updates are local and fast when the graph changes over time. We summarize our main contributions as follows.

- *Fast query processing*: We propose a fast algorithm to answer k-NN queries on large graphs, with query response time sub-linear in input graph size.
- *Dynamic updates*: The algorithm includes fast, incremental update procedures for handling additions or deletions of edges and vertices. Thus it also works with time-evolving graphs.
- *Disk-resident graphs*: Our method can operate when the graph resides entirely on disk. Moreover, it loads only a small and relevant portion of the graph into memory to answer a query or perform dynamic updates.

We demonstrate the effectiveness and efficiency of our method, w.r.t. query accuracy and response time, on large real-world graphs from various domains.

2 Preliminaries and Overview

Vertex-Proximity. We consider finding the k-NN's of a given vertex in a graph. To calculate the k-NN's of a vertex, one needs to define a distance metric between two vertices. A widely used proximity measure that is based on random walks is Personalized Page Rank (PPR). Given a restart vertex q and a parameter $\alpha \in (0, 1)$, consider the random walk with restart starting at vertex q, such that at any step when currently present at a vertex v, it chooses any of its neighbors with equal probability α/d_v, and returns to the restart vertex q with probability $(1-\alpha)$. The stationary probability at vertex u of the random walk with restart is defined as the PPR score of u with respect to the query vertex q. The PPR score is known to be robust under noise or small changes in the graph, in contrast to shortest paths, and favors existence of many short paths between vertices. Therefore, in this paper we consider the problem of finding the k-NN's of any given vertex in a graph, where proximity is measured by the PPR score.

Overview. The main challenge in answering a k-NN query is the computational overhead involved in simulating a random walk on a large graph that may not even fit in memory. We employ a divide-and-conquer principle to handle this challenge. We cluster the graph into relatively small vertex-clusters and decompose the problem into simulating intra-cluster and inter-cluster random walks.

For a subset S of vertices, conditioned on the event that the random walk is in S, the probability that it steps out of S is proportional to its conductance—the ratio of the weight of edges crossing S to the weight of all edges incident to

1. Pre-computation (Offline)
 a. Cluster the graph into low-conductance, possibly *overlapping* clusters. For each vertex v, we identify a unique cluster containing v and call it the parent of v. Store these clusters on one or more compute nodes.
 b. For each cluster, compute and store some auxiliary information relating to intra-cluster random walks, *independently* of other clusters.

2. Query processing (Online)
 a. Given a query vertex, identify the 'right' subset of clusters to consider. (If all the clusters are considered, then the final answer is exact.)
 b. Combine the auxiliary information of identified clusters to compute PPR scores.

3. Graph update processing (Online/Batch)
 a. Given an update (addition/deletion of one or more vertices/edges), identify the 'right' subset of clusters to update.
 b. Update the identified clusters and their auxiliary information.

Fig. 1. Main components of proposed framework

S. Thus a low-conductance cluster "holds" the random walk longer than a high-conductance cluster. This makes low conductance a natural choice for estimating quality of a cluster for our purposes. We allow the clusters to overlap since it is natural for a vertex to belong to multiple communities.

Consider a random walk with restart starting at q. Let the sequence of vertices the walk visits be $v_0, v_1, v_2, \ldots$. A vertex may appear several times in this sequence. The stationary distribution of this walk is the relative frequency with which different vertices appear in this walk. Now suppose $q \in S_i$. As the random walk steps through this sequence, it stays in cluster S_i for a while, then jumps to another cluster S_j. Next it stays in S_j for a while before jumping to yet another cluster, and so on. The clusters thus visited by the walk may also repeat. As a result, one can partition the walk into a sequence of contiguous blocks of vertices where each block represents a portion of the walk inside a cluster and consecutive blocks represent a jump from a cluster to another.

Now it is easy to describe how to simulate the random walk based on the clusters. For each cluster S_i and each "entry" vertex $v \in S_i$, one can compute the characteristics of the random walk inside S_i assuming it entered S_i through v. These characteristics include the probabilities with which it exits S_i to different "exit" vertices and the expected number of times it visits various vertices in S_i before exiting. Interestingly this information can be computed for each cluster S_i *independent* of other clusters. Our method pre-computes and stores this information for each cluster. At query time, it combines this information across different clusters to compute the desired PPR scores. Whenever the graph changes, due to addition/deletion of vertices/edges, it updates the relevant clusters and their information appropriately. Our overall framework is given in Figure 1.

3 Proposed Method

We propose ClusterRank, a method to address the following two problems.

P1) Given a large edge-weighted graph $\mathcal{G}$, a query vertex q in $\mathcal{G}$ and an integer k, find k vertices in $\mathcal{G}$ that have highest PPR scores w.r.t. q.

P2) Given a large edge-weighted graph $\mathcal{G}(t)$ at time t, a subset $D(t)$ of existing edges in $\mathcal{G}(t)$ and a set $A(t)$ of new edges, update the graph structure and the relevant auxiliary information to delete the edges in $D(t)$ and add the edges in $A(t)$, i.e., compute $\mathcal{G}(t+1) := (\mathcal{G}(t) \setminus D(t)) \cup A(t)$.

We next describe the components in Fig. 1 in detail. To simplify the presentation, we assume the graph is unweighted. Our techniques, however, extend to graphs with non-negative edge-weights and directed graphs.[5]

3.1 Pre-Computation

The pre-computation involves two steps which can be performed offline.

3.1.1. Clustering the graph. To distribute a graph across compute nodes, one can use any top-performing known graph partitioning algorithm [1, 8, 15, 19, 23, 22]. In this work, we use [1] which finds low-conductance clusters and allows clusters to overlap. We assign each vertex v to a unique cluster S containing v that contains the maximum number of v's neighbors. We call such a cluster the *parent* cluster of v. The notion of parent clusters is used while query processing.

3.1.2. Computing auxiliary information for clusters. Given the overlapping clustering computed as $\mathcal{S} = \{S_1, S_2, \ldots, S_p\}$, we next compute certain auxiliary information for each cluster $S_i \in \mathcal{S}$ independently of others clusters. Assume that the query vertex $q \notin S_i$ and assume that the random walk with restart enters S_i through a vertex $u \in S_i$. We simulate this random walk with restart till it exits cluster S_i. Suppose the random walk is at vertex $v \in S_i$. In one step, with probability $1 - \alpha$, the random walk restarts at q and hence exits S_i. With probability α, it picks a neighbor $w \in \Gamma(v)$ at random. Here $\Gamma(v)$ denotes the set of neighbors of v in $\mathcal{G}$. If $w \in S_i$, the random walk continues within S_i. If $w \notin S_i$, the random walk exits S_i to vertex w. The auxiliary information for each cluster S_i consists of two matrices, the *Count* matrix and the *Exit* matrix.

Count matrix. The count matrix C_i is an $|S_i| \times |S_i|$ matrix defined as follows. The entry $C_i(u, v)$, for $u, v \in S_i$, equals the expected number of times a random walk with restart (restarting at $q \notin S_i$) starting at u visits v before exiting S_i. The following lemma gives a closed-form expression for C_i. Let T_i be an $|S_i| \times |S_i|$ matrix that gives transition probabilities of a random walk within S_i without restart, i.e., for $u, v \in S_i$, let $T_i(u, v) = 1/|\Gamma(u)|$ if $v \in \Gamma(u)$ and 0 otherwise.

Lemma 1. *$C_i = (I - \alpha T_i)^{-1}$ where I is the $|S_i| \times |S_i|$ identity matrix.*

Proof. It is easy to see that C_i satisfies the following relation for all $u, v \in S_i$.

$$C_i(u,v) = \begin{cases} 1 + \alpha \displaystyle\sum_{w \in \Gamma(u) \cap S} T_i(u,w) C_i(w,v), & \text{if } v = u; \\ \alpha \displaystyle\sum_{w \in \Gamma(u) \cap S} T_i(u,w) C_i(w,v), & \text{otherwise.} \end{cases} \tag{1}$$

In matrix form, the above relation can be written as $C_i = I + \alpha T_i C_i$. □

[5] We modify directed graphs by adding a self-loop to each vertex, such that no vertex has out-degree zero. This ensures the random walk matrix remains a Markov chain.

Exit matrix. Let $B_i = \Gamma(S_i) \setminus S_i$ denote the set of vertices not in S_i that are adjacent to vertices in S_i. The exit matrix E_i is an $|S_i| \times (|B_i|+1)$ matrix defined as follows. The entry $E_i(u,b)$, for $u \in S_i$ and $b \in B_i$, is the probability that a random walk with restart (restarting at $q \notin S_i$) starting at u exits S_i while jumping to vertex $b \in B_i$. Since the random walk can exit S_i while jumping to a restart vertex q (assumed not to be in S_i), we have an additional column in E_i corresponding to q. Of course, we do not know the identity of the restart vertex q at the pre-computation phase. Therefore we treat q as a symbolic representative of the restart vertex. The entry $E_i(u,q)$, for $u \in S_i$, is the probability that the random walk exits S_i while jumping to the restart vertex q. The following lemma gives a closed-form expression for E_i. Let T_i^+ be an $|S_i| \times (|B_i|+1)$ matrix that gives exit probabilities of a random walk from vertices in S_i to vertices in $B_i \cup \{q\}$, i.e., for $u \in S_i$ and $b \in B_i$, let $T_i^+(u,b) = \alpha/|\Gamma(u)|$ if $b \in \Gamma(u)$ and 0 otherwise; and for $u \in S_i$, $T_i^+(u,q) = 1 - \alpha$.

Lemma 2. *$E_i = (I - \alpha T_i)^{-1} T_i^+ = C_i T_i^+$ where I is the identity matrix.*

Proof. It is easy to see that E_i satisfies the following relation $\forall u \in S_i$ and $\forall b \in B_i \cup \{q\}$.

$$E_i(u,b) = \begin{cases} 1 - \alpha + \alpha \displaystyle\sum_{w \in \Gamma(u) \cap S} T_i(u,w) E_i(w,v), & \text{if } b = q; \\ T_i^+(u,b) + \alpha \displaystyle\sum_{w \in \Gamma(u) \cap S} T_i(u,w) E_i(w,v), & \text{otherwise.} \end{cases} \tag{2}$$

In matrix form, the above relation can be written as $E_i = T_i^+ + \alpha T_i E_i$. □

There are a couple of ways in which one can compute matrices C_i and E_i for a cluster. One can directly use the closed-form expressions given above. In this case, computing auxiliary information for a cluster S_i containing s vertices and containing b vertices in the neighborhood $\Gamma(S_i) \setminus S_i$ involves, computing an inverse $(I - \alpha T_i)^{-1}$ of an $s \times s$ matrix and multiplying an $s \times s$ matrix and an $s \times (b+1)$ matrix. This takes a total of $O(s^3 + s^2 b)$ time using Gaussian elimination for inverting and textbook matrix products. One can reduce this time complexity by using Strassen's algorithm [24]. An alternative is to use the relations (1) and (2) to compute these matrices in an iterative fashion. This approach, however, is often found less effective than computing the matrix inverse.

3.2 Query Processing

The second component of ClusterRank deals with query answering, and consists of two steps: (1) updating the auxiliary information for those clusters that contain the query vertex, and (2) combining such information from a subset of "relevant" clusters to compute the final PPR scores.

3.2.1. Updating the matrices given a query vertex. Given a query vertex q, we first identify its unique parent cluster S_i. We then update the count matrix C_i and the exit matrix E_i to reflect the fact that the restart vertex now lies inside the cluster S_i. We remark that the count and exit matrices corresponding to any other cluster, say S_j with $j \neq i$, containing the query vertex are not updated.

Count matrix with a query vertex inside. Given a query vertex q and its parent cluster S_i, the count matrix C_i^q is an $|S_i| \times |S_i|$ matrix defined analogously. The entry $C_i^q(u,v)$, for $u, v \in S_i$, equals the expected number of times a random walk with restart (restarting at $q \in S_i$) starting at u visits v before exiting S_i. The following lemma gives a closed-form expression for C_i^q. Let Q_q be an $|S_i| \times |S_i|$ matrix with all entries in the column q equal to 1 and all other entries zero.

Lemma 3. *$C_i^q = (I - \alpha T_i - (1-\alpha)Q_q)^{-1}$ where I is the $|S_i| \times |S_i|$ identity.*

Proof. Recall that the random walk restarts at $q \in S_i$ at every step with probability $1-\alpha$. Therefore C_i^q satisfies the following relation for all $u, v \in S_i$.

$$C_i^q(u,v) = \begin{cases} 1 + (1-\alpha)C_i^q(q,v) + \alpha \sum\limits_{w \in \Gamma(u) \cap S} T_i(u,w) C_i^q(w,v), & \text{if } v = u; \\ (1-\alpha)C_i^q(q,v) + \alpha \sum\limits_{w \in \Gamma(u) \cap S} T_i(u,w) C_i^q(w,v), & \text{otherwise.} \end{cases} \tag{3}$$

In matrix form, the above relation becomes $C_i^q = I + (1-\alpha)Q_q C_i^q + \alpha T_i C_i^q$. □

Exit matrix given a query vertex inside. Given a query vertex $q \in S_i$, the exit matrix E_i^q is an $|S_i| \times |B_i|$ matrix defined analogously. The entry $E_i^q(u,b)$, for $u \in S_i$ and $b \in B_i$, is the probability that a random walk with restart (restarting at $q \in S_i$) starting at u exits S_i while jumping to vertex $b \in B_i$. Note that since the random walk does not exit S_i due to a restart, E_i^q has only $|B_i|$ columns.

The following lemma gives a closed-form expression for E_i^q. Let T_i^{+q} be an $|S_i| \times |B_i|$ matrix that gives exit probabilities of a random walk from vertices in S_i to vertices in B_i, i.e., for $u \in S_i$ and $b \in B_i$, let $T_i^+(u,b) = \alpha/|\Gamma(u)|$ if $b \in \Gamma(u)$ and 0 otherwise. This matrix is T_i^+ with the last column dropped.

Lemma 4. *$E_i^q = (I - \alpha T_i - (1-\alpha)Q_q)^{-1} T_i^{+q} = C_i^q T_i^{+q}$.*

Proof. Recall that the random walk restarts at $q \in S_i$ at every step with probability $1-\alpha$. Therefore E_i^q satisfies the following relation $\forall u \in S_i$ and $\forall b \in B_i$.

$$E_i^q(u,b) = T_i^{+q}(u,b) + (1-\alpha)E_i^q(q,b) + \alpha \sum_{w \in \Gamma(u) \cap S} T_i(u,w) E_i^q(w,v).$$

In matrix form, we can write the above as $E_i^q = T_i^{+q} + (1-\alpha)Q_q E_i^q + \alpha T_i E_i^q$. □

Updating the matrices using Sherman-Morrison formula. Observe, from Lemmas 1 and 3, that the expressions for C_i and C_i^q are quite similar—C_i is the inverse of a matrix and C_i^q is the inverse of the same matrix with $(1-\alpha)Q_q$ subtracted. Note also that $(1-\alpha)Q_q$ is a rank-1 matrix. We can update the inverse of a matrix efficiently when the matrix undergoes such a low-rank update. To this end, we first quote a well-known lemma.

Lemma 5 (Sherman-Morrison-Woodbury [27]). *Let n and k be any positive integers, $A \in \Re^{n \times n}, U \in \Re^{n \times k}, \Sigma \in \Re^{k \times k}, V \in \Re^{k \times n}$ be any matrices:*

$$(A + U\Sigma V)^{-1} = A^{-1} - A^{-1}U(\Sigma^{-1} + VA^{-1}U)^{-1}VA^{-1}.$$

Note that $U\Sigma V$ is a rank-k matrix. Thus after updating A with a rank-k matrix, its inverse can be computed from A^{-1} by doing 4 multiplications of $n \times n$ and $n \times k$ matrices, 2 multiplications of $n \times k$ and $k \times k$ matrices and 1 inverse of a $k \times k$ matrix. Thus overall time is $O(n^2 k)$ since $k \leq n$. This can be much more efficient than computing the inverse of an $n \times n$ matrix from scratch, especially if k is much smaller than n. If $k = 1$, the above formula reduces to what is commonly known as Sherman-Morrison formula. We refer to the formula in the above lemma as the SMW formula in the remainder of the text.

To use this approach, we have to express the rank-1 matrix $(1-\alpha)Q_q$ as $U\Sigma V$ where Σ is a 1×1 matrix, i.e., a scalar. This can be done simply by setting $\Sigma = (1-\alpha)\sqrt{|S_i|}$, U to be an $|S_i|$-size column vector with all entries $1/\sqrt{|S_i|}$ and V to be an $|S_i|$-size row vector with all entries 0 except the entry corresponding to q equal to 1. Thus C_i^q can be computed from C_i in $O(|S_i|^2)$ time. Similarly, E_i^q can be computed from C_i and E_i in $O(|S_i|^2 + |S_i||B_i|)$ time.

To simplify the notation in the following discussion, we let $\hat{C}_i$ (resp. $\hat{E}_i$) denote C_i^q (resp. E_i^q) if S_i is the parent cluster of q, and C_i (resp. E_i) otherwise.

3.2.2. Computing the PPR scores. Recall that to compute the PPR scores, our method decomposes the random walk with restart starting from the query vertex q into intra-cluster and inter-cluster random walks. Since the information about intra-cluster random walks is already pre-computed (or appropriately updated for the parent cluster of the query vertex), we next compute the necessary information about the inter-cluster random walk. As a first step, we identify the clusters "relevant" for answering the k-NN query for q. If we want to compute PPR scores exactly, we label all the clusters as relevant. Working with all the clusters to answer a query, however, leads to excessive query response time. It turns out that one can reduce the query response time significantly by limiting the number of relevant clusters. We employ two heuristics called *1-hop* and *2-hop* to limit the relevant clusters. In the former, we label a cluster S as relevant if and only if $q \in S$. In the latter, we label a cluster S as relevant if and only if either $q \in S$ or S is the parent cluster of some vertex $b \in B_i = \Gamma(S_i) \setminus S_i$ for some cluster S_i such that $q \in S_i$. Intuitively, these heuristics quickly identify the vertices that are expected to have high PPR scores w.r.t. q.

Suppose $\mathcal{S}_q$ is the set of relevant clusters. Let $\cup\mathcal{S}_q$ denote the union of these clusters. Recall that $B_i = \Gamma(S_i) \setminus S_i$ denotes the set of vertices which the random walk inside S_i may jump to while exiting S_i. Now let $B_q = ((\cup\mathcal{S}_q) \cap (\cup_{S_i \in \mathcal{S}_q} B_i)) \cup \{q\}$. As the number of vertices in B_q relates to the efficiency of our method, we explicitly limit $|B_q|$; we fix a parameter β, and while using either 1-hop or 2-hop heuristic, we continue labeling the clusters relevant as long as $|B_q|$ does not exceed β or all the clusters according to the heuristic are labeled relevant.

Computing the inter-cluster random walk matrix. After identifying the relevant clusters $\mathcal{S}_q$, we gather their auxiliary information to compute the inter-cluster random walk matrix. Recall that when the random walk with restart enters $u \in S_i \in \mathcal{S}_q$, it exits S_i while jumping to some vertex $b \in B_i \cup \{q\}$ (or $b \in B_i$ if S_i is the parent cluster of q). The probability of this event is exactly given by

$\hat{E}_i(u, b)$. Clearly, this vertex b can belong to multiple clusters. When the random walk jumps to b, we assume that it enters the parent cluster of b.

Thus we can think of inter-cluster jumps as a random walk on the vertices in B_q. Whenever the random walk jumps to a vertex $b \in B_i \setminus \cup\mathcal{S}_q$ that is not in the relevant clusters, we assume that the random walk jumps back to q. The transition matrix (of dimensions $|B_q| \times |B_q|$) of this walk is as follows. For any $b_1, b_2 \in B_q$, the probability that this random walk jumps from b_1 to b_2 is

$$M_q(b_1, b_2) = \begin{cases} \hat{E}_i(b_1, b_2), & \text{if } b_2 \neq q, S_i \text{ is theparent cluster of } b_1; \\ 1 - \sum_{b \in B_q \setminus \{q\}} M_q(b_1, b), & \text{if } b_2 = q. \end{cases} \tag{4}$$

We compute M_q from the auxiliary information stored (or appropriately updated) for the relevant clusters. Recall that the random walk (or the corresponding Markov chain) is called *ergodic* if it is possible to go from every state to every other state (not necessarily in one move), and if the walk is aperiodic. Now we can assume that the given graph $\mathcal{G}$ is connected without loss of generality.[6] Thus, if we label all clusters as relevant, the resulting Markov chain M_q is ergodic (under very mild assumptions satisfied by large real-world graph topologies). Also, from the definition of 1-hop or 2-hop heuristics, the resulting Markov chain M_q is still ergodic even if we use these heuristics.

From the standard theorem of ergodic chains [9], we conclude that there is a *unique* probability row-vector $\mu \in \Re^{|B_q|}$ such that $\mu M_q = \mu$. This vector gives the expected fraction of steps the random walk spends at any vertex $b \in B_q$. This vector can be computed either by doing repeated multiplications of M_q with the starting probability distribution (which is 1 at the coordinate q and 0 elsewhere); or by computing the top eigenvector of $I - M_q^\top$ corresponding to eigenvalue 1. The eigenvector computation can be done in time $O(|B_q|^3)$.

We now "lift" this random walk back to the random walk with restart on the union of the relevant clusters $\cup\mathcal{S}_q$. Since a cluster $S_i \in \mathcal{S}_q$, the value $\hat{C}_i(u, v)$ gives the expected number of times the random walk with restart (starting at q) visits $v \in S_i$ before exiting S_i. Therefore, for a vertex $v \in \cup\mathcal{S}_q$, the quantity

$$\pi_v = \sum_{S_i \in \mathcal{S}_q : v \in S_i} \; \sum_{b \in B_q : S_i \text{ parent of } v} \mu_b \hat{C}_i(b, v) \tag{5}$$

gives the expected number of times the random walk with restart visits $v \in \cup\mathcal{S}_q$ between consecutive inter-cluster jumps. Scaling these values so that they sum up to 1, gives the fraction of steps the random walk visits $v \in \cup\mathcal{S}_q$, i.e., $\hat{\pi}_v = \frac{\pi_v}{\sum_{u \in \cup\mathcal{S}_q} \pi_u}$. The k-NN query can then be answered by identifying k vertices with the highest values of $\hat{\pi}_v$. The following theorem is now evident.

Theorem 1. *If we label all the clusters as relevant, the computed values $\{\hat{\pi}_v \mid v \in \mathcal{G}\}$ equal the* exact PPR *values w.r.t. the query vertex q.*[7]

The k-NN query is then answered by top k vertices with the highest $\hat{\pi}_v$.

[6] If $\mathcal{G}$ is not connected, we focus on the connected component containing q.

[7] For directed graphs, only vertices reachable from q by a directed path get non-zero PPR values, i.e. $\{\hat{\pi}_v > 0 \mid q \rightsquigarrow v \in \mathcal{G}\}$.

3.3 Dynamic Updates

To simplify the presentation, we describe how to handle addition of a single edge $e = \{u, v\}$. When an edge is added, the transition probability matrices T_i and T_i^+ for some clusters S_i are changed, resulting in the change of C_i and E_i according to Lemmas 1 and 2. The key observation here is that the changes in T_i and T_i^+ are low-rank. Therefore, new C_i and E_i can be computed from their old versions using the SMW formula. Let $\mathcal{S}(u)$ be the set of clusters containing u and $\mathcal{S}(v)$ containing v. We consider several cases.

Case 1: $\mathcal{S}(\mathbf{u}) = \mathcal{S}(\mathbf{v}) = \emptyset$. This case arises when both vertices u and v are new vertices. In this case, we add both these vertices to a cluster S_i with the smallest size. We also designate S_i as the parent cluster of both u and v. Note that the edge e forms a disconnected component in S_i. Therefore, the matrices C_i and E_i can be computed directly without resorting to the SMW formula.

Consider a random walk with restart (restarting at q) starting at u. Since the random walk restarts with probability $1 - \alpha$, it is easy to see that the expected number of times the random walk visits u before exiting $\{u, v\}$ is $\frac{1}{1-\alpha^2}$ and the expected number of times the random walk visits v before exiting $\{u, v\}$ is $\frac{\alpha}{1-\alpha^2}$. It is now easy to observe that if the edge $e = \{u, v\}$ is added to cluster S_i, it's count matrix can be computed from the original count matrix C_i as on the left.

$$\left[\begin{array}{c|cc} & 0 & 0 \\ C_i & \vdots & \vdots \\ & 0 & 0 \\ \hline 0 \dots 0 & \frac{1}{1-\alpha^2} & \frac{\alpha}{1-\alpha^2} \\ 0 \dots 0 & \frac{\alpha}{1-\alpha^2} & \frac{1}{1-\alpha^2} \end{array}\right]$$

We then use Lemma 2 to compute the new E_i as $C_i T_i^+$ where the new T_i^+ is computed as $\left[\begin{array}{c} T_i^+ \\ \hline 0 \dots 0 \; \alpha \\ 0 \dots 0 \; \alpha \end{array}\right]$.

Case 2: $\mathcal{S}(\mathbf{v}) = \emptyset$. In this case, vertex v is a new vertex. We add vertex v to each cluster $S_i \in \mathcal{S}(u)$ and designate some cluster picked arbitrarily among these as the parent cluster of v. We now use the SMW formula along with Lemma 1 to compute the new count matrix C_i. The probability transition matrix T_i is updated as follows. Let d_u be the degree of u before adding v. Add a new row and a new column both corresponding to vertex v to T_i and add matrix A_i of the same dimensions where A_i has all entries zero except $A_i(u, w) = -1/d_u(d_u + 1)$ for all $w \in S_i \cap \Gamma(u)$, $A_i(u, v) = 1/(d_u + 1)$, $A_i(v, u) = 1$. Since A_i has only two non-zero rows that are linearly-independent, it has rank 2. Thus we compute an SVD decomposition $A_i = U\Sigma V$ where Σ is a 2×2 matrix and update C_i using Lemma 5. We again use Lemma 2 to compute the new matrix E_i as $C_i T_i^+$ where the matrix T_i^+ is updated by adding a new row corresponding to v with all entries zero and adding a matrix A_i^+ of the same dimensions. Here A_i^+ is a matrix with all zero entries except $A_i^+(u, b) = -\alpha/d_u(d_u + 1)$ for all $b \in B_i \cap \Gamma(u)$ and $A_i(v, q) = 1 - \alpha$.

Case 3: $\mathbf{u} \in \mathbf{S_i}$ and $\mathbf{v} \notin \mathbf{S_i}$. We again use the SMW formula along with Lemma 1 to compute the new count matrix C_i. The probability transition matrix T_i is now updated as follows. Let d_u be the degree of u before adding edge $e = \{u, v\}$. The matrix T_i is updated by adding a matrix A_i of the same dimensions where A_i has all entries zero except $A_i(u, w) = -1/d_u(d_u + 1)$ for all $w \in S_i \cap \Gamma(u)$. Since A_i has only one non-zero row, it has rank 1. Thus again we compute an SVD decomposition $A_i = U\Sigma V$ where Σ is a scalar and update C_i using Lemma 5.

We use Lemma 2 to compute the new E_i as $C_i T_i^+$ where T_i^+ is updated as follows. If $v \notin B_i$ currently, we add a column corresponding to v with all entries zero. Next we update T_i^+ as $T_i^+ + A_i^+$ where A_i^+ is a matrix with all zero entries except $A_i^+(u,b) = -\alpha/d_u(d_u+1)$ for all $b \in B_i \cap \Gamma(u)$ and $A_i^+(u,v) = \alpha/(d_v+1)$. The case where $u \in S_i$ and $v \notin S_i$ is analogous and is omitted.

Case 4: $\mathbf{u}, \mathbf{v} \in \mathbf{S_i}$. The probability transition matrix T_i is now updated as follows. Let d_u be the degree of u and d_v be the degree of v before adding edge $e = \{u, v\}$. The matrix T_i is updated by adding a matrix A_i of the same dimensions where A_i has all entries zero except $A_i(u,w) = -1/d_u(d_u+1)$ for all $w \in S_i \cap \Gamma(u)$, $A_i(u,v) = 1/(d_u+1)$, $A_i(v,w) = -1/d_v(d_v+1)$ for all $w \in S_i \cap \Gamma(v)$, $A_i(v,u) = 1/(d_v+1)$. Since A_i has only two non-zero rows that are linearly-independent, it has rank 2. Thus again we compute an SVD decomposition $A_i = U\Sigma V$ where Σ is a 2×2 matrix and update C_i using Lemma 5.

We use Lemma 2 to compute the new E_i as $C_i T_i^+$ where T_i^+ is updated by adding a matrix A_i^+ of the same dimensions. Here A_i^+ is a matrix with all zero entries except $A_i^+(u,b) = -\alpha/d_u(d_u+1)$ for all $b \in B_i \cap \Gamma(u)$ and $A_i^+(v,b) = -\alpha/d_v(d_v+1)$ for all $b \in B_i \cap \Gamma(v)$.

4 Empirical Study

We evaluate our method, with respect to accuracy and efficiency, on both synthetic and real-world graphs. We first give dataset description including synthetic data generation and follow with experiment results.[8]

Synthetic data generation. Our graph generation algorithm is based on the planted partitions model [6]. Simply put, given the desired number of vertices in each partition we split the adjacency matrix into blocks defined by the partitioning. For each block B_{ij}, the user provides a probability p_{ij}. Using a random process based on this probability we assign a 1, i.e. an edge, to each entry in each block, and 0 otherwise. In other words, p_{ij} specifies the density of each block.

Using the above planted partitions model, we simulated a graph of 300K vertices, with 100 partitions of equal size. We set $p_{ii} = 10^{-3}$ and $p_i = \sum_{j,j\neq i} p_{ij} = 10^{-5}$, which yielded 909,333 edges in the graph.

Real datasets. Our real graph datasets come from diverse domains such as social, Web, and co-authorship networks, and vary in size from 1 million edges to more than 20 million edges. We give a summary of our datasets in Table 1.

Table 1. Graph datasets (E: #edges, N: #vertices, real graph data source: http://snap.stanford.edu/data) — C: #clusters and median conductance ϕ and size.

Dataset	E	N	Description	C	med. ϕ	med. size
Synthetic	300K	909K	Planted partitions [6]	100	0.0210	3050
Web	1100K	325K	http://nd.edu links	2793	0.0625	31
Amazon	900K	262K	Product co-purchases	3739	0.1385	17
DBLP	1100K	329K	Co-authorships	4670	0.2117	27
Live Journal	21500K	2700K	Friendships	15252	0.5500	43

[8] All experiments are performed on a 3-CPU 2.8 GHz AMD Opteron 854 server with 32GB RAM. The fly-back probability α for random walks is set to 0.15.

4.1 Pre-computation

The first phase involves clustering a given graph and then computing auxiliary information, i.e., the count and exit matrices C and E for each cluster.

There are several top-performing algorithms known for graph clustering such as [1, 8, 15, 19]. We performed experiments using both METIS [15] and the Andersen *et. al.* algorithm [1]. For moderate sized graphs (e.g., with 1.1 million edges), the qualities of query results obtained with either of the clustering algorithms were comparable, both in running time and accuracy. However, for large graphs (e.g., Live Journal with 21.5 million edges), METIS could not complete the clustering computation while Andersen *et. al.* algorithm was able to compute a clustering—it took about 35 seconds to compute each of the 15,000 clusters.

Table 1 shows the median conductance and size of the clusters found in each dataset. As shown in [17], we observe that good conductance clusters are often of small size. Moreover, graphs from different domains cluster differently, where lower conductance implies higher quality clusters.

In Figure 2, we show the pre-computation time for our graphs which consists of two parts; the first (blue) bars show the average time to extract a single cluster, and the second (red) bars give the average time to compute its corresponding C and E matrices. Note that as each graph clustered into different number of clusters, and hence we show the average time per cluster.

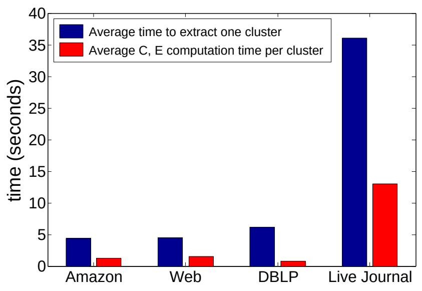

Fig. 2. Average time per cluster for two phases of pre-computation: (1) clustering and (2) C, E computation, for real graphs.

4.2 Query Processing

After pre-computation, our method is ready to process queries. In order to measure performance, we conducted experiments with 100 randomly chosen query vertices from each graph. We report average running time and average accuracy on all graphs. To compute accuracy, we need the "true" PPR scores. Thus, we also compute the exact PPR (EXPPR) scores using power-iterations [10].

One of the measures for accuracy is "precision at k" which can be defined as $|T_k \cap \hat{T}_k|/k \in [0, 1]$, where T_k and $\hat{T}_k$ denote the sets of top-k vertices using the exact and the test algorithm, respectively. However, precision can be excessively severe. In many real graphs, ties and near-ties in PPR scores are very common. In such a case, we would like to say that the test algorithm works well if the "true" scores of $\hat{T}_k$ are large. Therefore we use the Relative Average Goodness (RAG) at k which is defined as $\text{RAG}(k) = \frac{\sum_{v \in \hat{T}_k} p(v)}{\sum_{v \in T_k} p(v)} \in [0, 1]$ where $p(v)$ denotes the "true" PPR score of vertex v w.r.t. the query vertex.

Figure 3 shows the average RAG accuracy versus response time achieved by CLUSTERRANK for all four real-world graphs. EXPPR response time is also shown

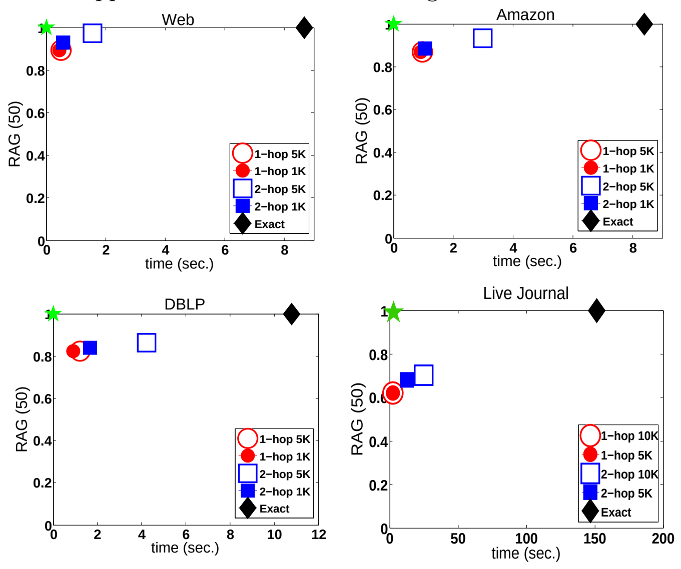

Fig. 3. Average accuracy (RAG(50)) vs. response time (sec.) for (from left to right) Web, Amazon, DBLP and Live Journal of ClusterRank *1-hop* and *2-hop* heuristics compared to Exppr. The optimal point is depicted with a star.

with an RAG score of 1. For the Web graph, both *1-hop* and *2-hop* average accuracy is quite close to that of the exact algorithm (0.90 and 0.97, respectively). As one might expect, the best accuracy is achieved using *2-hop* with β=5K (i.e., max. number of boundary vertices) since in that case more clusters are considered as relevant. Notice the results are similar for other graphs. We show the accuracy and response times on our synthetic graph on the right, which suggests that high accuracy is achieved for graphs with well-pronounced clusters.

	2-hop	*1-hop*
β =5K	0.9986 / 5.12	0.9865 / 2.18
β =1K	0.9892 / 2.86	0.9865 / 2.12

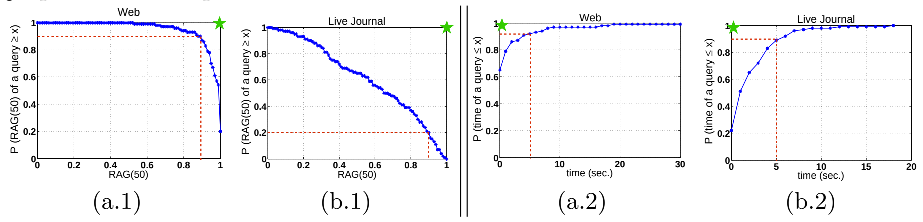

Fig. 4. NCDF distribution of RAG(50) scores for (a.1) Web, (b.1) Live Journal; and CDF distribution of query response times for (a.2) Web, (b.2) Live Journal. (using the *1-hop* heuristic and β=5K). Our method performs better on Web graph than on Live Journal, possibly due to higher quality clusters. Optimal point is depicted with a star.

In Figure 4, we show the distribution of (a) accuracy scores and (b) running times of all the 100 queries in Web and Live Journal. The ideal point is also marked with a star on each figure. We observe that around 80% of the queries in Web (and around 20% in Live Journal) have an accuracy more than 0.9. Also, 90% of the queries take less than 5 seconds in both graphs.

We further study the performance of ClusterRank on increasing graph sizes. We first merge clusters from our largest graph Live Journal to build a connected graph $\mathcal{G}_{1/2M}$ with half a million edges, and keep growing the number of clusters to obtain a set of increasingly larger graphs, $\{\mathcal{G}_{1M}, \mathcal{G}_{2M}, \ldots, \mathcal{G}_{21M}\}$. As before, we conduct experiments on 100 randomly chosen query vertices from $\mathcal{G}_{1/2M}$ and keep the same query set for the larger graphs to ensure that the query vertices exist in all graphs.

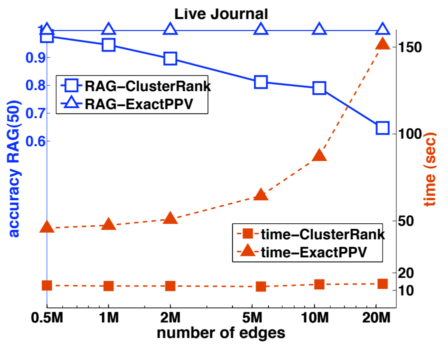

Fig. 5. Accuracy and response time for ClusterRank (squares) and Exppr (triangles) with increasing graph size.

Figure 5 shows both RAG accuracy and response time versus graph size for ClusterRank and Exppr. ClusterRank's accuracy remains ≈ 0.70 when the graph becomes more than $40\times$ larger. Moreover, its response time stays almost constant at around 13 seconds across graphs with increasing size, while Exppr's response time grows up to 150 seconds following a quadratic trend.

Next, we analyze our results qualitatively. We build the DBLP graph for years 2000-2007, and run our method on 100 randomly chosen authors. We list the top-proximity authors we found to two example authors.[9] Bold-faced authors are found to be past or future collaborators, while others are highly related with overlapping research interests (respectively, parallel computing and biomedical imaging).

'Christoph_Zenger'	'Shigehiko_Katsuragawa'
'Hans-Joachim_Bungartz'	'Joo_Kooi_Tan'
'Ralf-Peter_Mundani'	'Yoshinori_Otsuka'
'Ralf_Ebner'	'Feng_Li'
'Tobias_Weinzierl'	'Masahito_Aoyama'
'Anton_Frank'	'Shusuke_Sone'
'Ioan_Lucian_Muntean'	'Takashi_Shinomiya'
'Thomas_Gerstner'	**'Heber_MacMahon'**
'Clemens_Simmer'	'Junji_Shiraishi'
'Dirk_Meetschen'	**'Roger_Engelmann'**
'Susanne_Crewell'	'Kenya_Murase'

4.3 Dynamic Updates

To study the performance of ClusterRank on dynamic updates, we use DBLP which is a time-varying graph by years. First, we build a DBLP co-authorship graph of 500K edges which spans from 1959 to 2001. Then, we perform updates to our clusters by introducing the next 1K edges in time. Note that some new edges also introduce new vertices to the graph.

Figure 6 (left) shows the distribution of the number of clusters affected per edge for the 1K new edges added, and (right) the distribution of update times. We note that more than 90% of the new edges cause fewer than 5 clusters to be updated. Moreover, 90% of the updates take less than 100 seconds, including reading/writing of the C and E matrices of the affected clusters from/to the disk.

[9] Note that direct neighbors, i.e. co-authors, are omitted from the top list as they constitute trivial nearest neighbors.

Finally Table 2 shows ClusterRank's accuracy on the time-evolving DBLP graph; i.e., after the addition of (a) 1K and (b) 500K new edges to it, which initially had 500K edges. We notice that the accuracy remains quite stable over the course of the changes to the graph, even after when the graph size doubles.

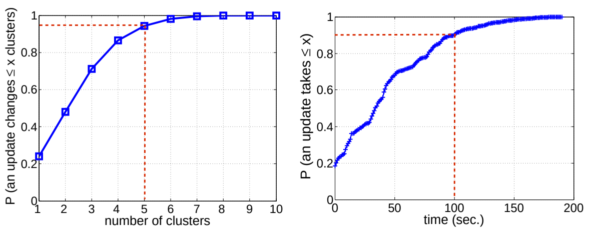

Fig. 6. Distributions of (left) number of clusters affected and (right) affected cluster update times, per new edge.

Table 2. Accuracy on DBLP with 500K, and after it grows to 501K, and 1M edges.

	DBLP(500K)		DBLP(501K)		DBLP(1M)	
	2-hop	*1-hop*	*2-hop*	*1-hop*	*2-hop*	*1-hop*
β =5K	0.8887	0.8241	0.8717	0.8124	0.8840	0.8375
β =1K	0.8565	0.8210	0.8446	0.8095	0.8583	0.8327

5 Related Work

Scalable and efficient algorithms for exact as well as approximate computation of Page Rank (PR) scores in graphs has been studied widely [5, 7, 14, 16, 18, 20]. On the other hand, computing the *Personalized* Page Rank (PPR) scores for a given vertex, which is the central topic of our paper, is a considerably more general and harder problem than computing PR scores. The reason is that PR scores of vertices in a graph are stationary probabilities over a network-wide random walk; hence, there is a *single* (global) PR score of each vertex. On the contrary, PPR scores change as a function of the start vertex, and thus are a significant generalization of PR scores.

Designing efficient algorithms to compute PPR scores has been an active topic of relatively recent research due to its many applications including link-prediction, proximity tracking in social networks and personalized web-search. Tong *et al.* [25] develop fast methods for computing PPR scores. They exploit community structure by graph partitioning and correlations among partitions by low-rank approximation. However, their method is tuned for static graphs and does not address dynamic updates. The later work by Tong *et al.* [26] is the one which is closely related to our work; here, the authors consider proximity tracking in time-evolving *bipartite* graphs and develop matrix algebraic algorithms for efficiently answering proximity queries. The assumption that the graph is bipartite, and further, the assumption that one of its partitions is of a small size is critical to their design. For instance, these assumptions play a key role in constructing a low-rank approximation of the graph adjacency matrix, which is then perturbed to account for edge additions over time.

Sarkar and Moore [21] develop fast external memory algorithms for computing PPR scores on disk-resident graphs. They focus on suitable cluster representations in order to optimize disk accesses for minimizing query-time latency for *static* graphs. MapReduce based methods [3] optimized for disk-resident graphs also cannot deal with dynamic graphs.

The development of efficient algorithms based on linear algebraic techniques and segmented random-walks for fast computation of PPR scores on *static* graphs has also been the subject of several works. Haveliwala [12] pre-compute multiple importance scores w.r.t. various topics for each Web page towards personalization; at query time, these scores are combined to form a context-specific, composite PPR scores. Jeh and Widom [13] propose efficient algorithms to compute "partial vectors" encoding personalized views, which are incrementally used to construct the fully personalized view at query time. Fogaras *et al.* [10] use simulated random walk segments to approximate PPR scores by stitching the walk segments to form longer walks. Chakrabarti *et al.* [4, 11] pre-compute random walk "fingerprints" for a small fraction of the so-called hub vertices. At query time, an "active" subgraph bounded by hubs is identified where PPR scores are estimated by iterative PPV decompositions.

Most related to ours is the work by Bahmani *et al.* [2], which also uses a divide-and-conquer approach: pre-computation (random walk segments) and query-time combination of intermediate results (random walk using segments), with fast query and update times. The main difference is their underlying infrastructure; [2] needs distributed shared memory for its employed random-access Monte Carlo method, while we can work with fully distributed commodity systems—once the graph is partitioned, the compute nodes operate independently, and a dedicated node combines results only from relevant nodes.

6 Conclusion

We propose ClusterRank, an efficient method for answering PPR-based k-NN queries in large time-evolving graphs. Our method addresses three major challenges associated with this problem: *(1) fast k-NN queries*; at query time, we operate on a small subset of clusters and their pre-computed information, and achieve a response time *sub-linear* in the size of the graph, *(2) efficient incremental dynamic updates*; thanks to our divide-and-conquer approach, addition or deletion of an edge/vertex triggers the update of only a small subset of clusters, which involves at most rank-2 updates to their pre-computed information, and *(3) spilling to disk*; as both query processing and dynamic updates operate on subset of clusters, only a small fraction of the graph is loaded into memory at all times while the rest sits on disk. As such, the modular design of our approach is a natural way to handle both large and time-evolving graphs simultaneously.

Acknowledgments

The authors would like to thank Christos Faloutsos for valuable discussions and the anonymous reviewers for their useful feedback. This material is based upon work supported by an ARO Young Investigator Program grant under Contract No. W911NF-14-1-0029, an ONR SBIR grant under Contract No. N00014-14-P-1155, an R&D gift from Northrop Grumman, and Stony Brook University Office of Vice President for Research. Any findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the official policies or views of the funding parties.

References

1. R. Andersen, F. R. K. Chung, K. J. Lang. Local graph partitioning using pagerank vectors. FOCS 2006.
2. B. Bahmani, A. Chowdhury, A. Goel. Fast Incremental and Personalized PageRank. 4 (3), pages 173-184, PVLDB, 2010.
3. B. Bahmani, K. Chakrabarti, D. Xin. Fast personalized PageRank on MapReduce. SIGMOD 2011.
4. S. Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. WWW 2007.
5. Y.-Y. Chen, Q. Gan, and T. Suel. Local methods for estimating pagerank values. CIKM 2004.
6. A. Condon and R. M. Karp. Algorithms for graph partitioning on the planted partition model. *Random Struct. Algorithms*, 18 (2), 2001.
7. A. Das Sarma, S. Gollapudi, R. Panigrahy. Estimating pagerank on graph streams. PODS 2008.
8. C. H. Q. Ding, X. He, H. Zha, M. Gu, H. D. Simon. A min-max cut algorithm for graph partitioning and data clustering. ICDM 2001.
9. W. Feller. An Introduction to Probability Theory and Its Applications, Wiley, 1971.
10. D. Fogaras and B. Rácz. Towards scaling fully personalized pagerank. Algorithms and Models for the Web-Graph, LNCS 2004.
11. M. Gupta, A. Pathak, and S. Chakrabarti. Fast algorithms for top-k personalized pagerank queries. WWW 2008.
12. T. H. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. WWW 2002.
13. G. Jeh and J. Widom. Scaling personalized web search. WWW 2003.
14. S. Kamvar, T. Haveliwala, G. Golub. Adaptive methods for the computation of pagerank. Linear Algebra Appl. 386, 51-65, 2004.
15. G. Karypis and V. Kumar Multilevel algorithms for multi-constraint graph partitioning. *Supercomputing*, 1-13, 1998.
16. A. N. Langville and C. D. Meyer. Updating pagerank with iterative aggregation. WWW Alt. 2004.
17. J. Leskovec, K. J. Lang, A. Dasgupta, M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6(1)* 2009.
18. F. McSherry. A uniform approach to accelerated pagerank computation. WWW 2005.
19. A. Y. Ng, M. I. Jordan, Y. Weiss On spectral clustering: Analysis and an algorithm. NIPS 2001.
20. L. Page, S. Brin, R. Motwani, T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.
21. P. Sarkar and A. W. Moore. Fast nearest-neighbor search in disk-resident graphs. KDD 2010.
22. S. E. Schaeffer. Graph clustering. *Computer Science Review*, I:27–64, 2007.
23. I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. pages 1222–1230, KDD 2012.
24. V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.
25. H. Tong, C. Faloutsos, and J.-Y. Pan. Fast random walk with restart and its applications. ICDM 2006.
26. H. Tong, S. Papadimitriou, P. S. Yu, C. Faloutsos. Proximity Tracking on Time-Evolving Bipartite Graphs. SDM 2008.
27. M. A. Woodbury. Inverting modified matrices. *Memorandum, Statistical Research Group*, 42, 1950.