Fast Nearest-neighbor Search in Disk-resident Graphs

Purnamrita Sarkar   Andrew W. Moore

February 2010
CMU-ML-10-100

ML

MACHINE LEARNING
D E P A R T M E N T

Carnegie Mellon

Fast Nearest-neighbor Search in Disk-resident Graphs

Purnamrita Sarkar   Andrew W. Moore

February 5, 2010
CMU-ML-10-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Link prediction, personalized graph search, fraud detection, and many such graph mining problems revolve around the computation of the most "similar" k nodes to a given query node. One widely used class of similarity measures is based on random walks on graphs, e.g., personalized pagerank, hitting and commute times, and simrank. There are two fundamental problems associated with these measures. First, existing online algorithms typically examine the local neighborhood of the query node which can become significantly slower whenever high-degree nodes are encountered (a common phenomenon in real-world graphs). We prove that turning high degree nodes into sinks results in only a small approximation error, while greatly improving running times. The second problem is that of computing similarities at query time when the graph is too large to be memory-resident. The obvious solution is to split the graph into clusters of nodes and store each cluster on a disk page; ideally random walks will rarely cross cluster boundaries and cause page-faults. Our contributions here are twofold: (a) we present an efficient deterministic algorithm to find the k closest neighbors (in terms of personalized pagerank) of any query node in such a clustered graph, and (b) we develop a clustering algorithm (RWDISK) that uses only sequential sweeps over data files. Empirical results on several large publicly available graphs like DBLP, Citeseer and Live-Journal (~ 90 M edges) demonstrate that turning high degree nodes into sinks not only improves running time of RWDISK by a factor of 3 but also boosts link prediction accuracy by a factor of 4 on average. We also show that RWDISK returns more desirable (high conductance and small size) clusters than the popular clustering algorithm METIS, while requiring much less memory. Finally our deterministic algorithm for computing nearest neighbors incurs far fewer page-faults (factor of 5) than actually simulating random walks.

1   Introduction

A number of important real world applications (e.g. collaborative filtering in recommender networks, link prediction in social networks, fraud detection, and personalized graph search techniques) rely on finding nearest neighbors in large graphs, where "nearness" is defined using graph-theoretic measure of similarity. A widely applied class of measures are based on random walks on graphs [5, 15, 13]; examples include personalized pagerank, hitting time and commute time, simrank, etc. Yet, there are limitations to what we can do when graphs become enormous.

Some algorithms, such as streaming algorithms [18], must make passes over the entire dataset to answer any query; this can be prohibitively expensive in online settings. Others perform clever preprocessing so that queries can be answered efficiently [9, 17]. However these algorithms store information which can be used for computing a *specific* similarity measure (e.g., personalized pagerank for [9]). This paper introduces analysis and algorithms which try to address the scalability problem in a generalizable way: not specific to one kind of graph partitioning nor one specific proximity measure.

Another broad class of algorithms estimate the similarity between the query node and other nodes by examining local neighborhoods around the query node [3, 2, 5, 17, 20]. The intuition is that in order to compute nearest neighbors, hopefully one would not need to look too far away from the query node. However, one fundamental computational issue with these techniques is the presence of very high degree nodes in the network. These techniques rely on updating one node's value by combining that of its neighbors; whenever a high degree node is encountered these algorithms have to examine a much larger neighborhood leading to severely degraded performance. Unfortunately, real-world graphs contain such high-degree nodes which, though few in number, are easily reachable from other nodes and hence are often encountered in random walks. Our first contribution is a simple transform of the graph that can mitigate the damage while having a provably bounded impact on accuracy. Indeed, we show that they *improve* accuracy in certain tasks like link prediction.

Another problem linked to large graphs is that algorithms can no longer assume that the entire graph can be stored in memory. In some cases, clever graph compression techniques can be applied to fit the graphs into main memory, but there are at least three settings where this might not work. First, social networks are far less compressible than Web graphs [7]. Second, decompression might lead to an unacceptable increase in query response time. Third, even if a graph could be compressed down to a gigabyte (comfortable main memory size in 2009) it is undesirable to keep it in memory on a machine which is running other applications, and in which there are occasional user queries to the graph. A good example of this third case is the problem of searching personal information networks [6], which integrates the user's personal information with information from the Web and hence needs to be performed on the user's own machine for privacy preservation [8].

Is there an intuitive representation of a disk-resident graph such that any random walk based measure can be easily computed from this representation? The obvious solution is to split the graph into clusters of nodes and store each cluster on a disk page; ideally random walks will rarely cross cluster boundaries and cause page-faults. This *clustered representation* can be used to quickly simulate random walks from any graph node, and by extension, any similarity measure based on random walks. Still, while simulations are computationally cheap, they have a lot of variation, and for some real-world graphs lacking well-defined clusters, they often lead to many page-faults.

We propose a *deterministic* local algorithm guaranteed to return nearest neighbors in personalized pagerank from the disk-resident clustered graph. The same idea can also be used for computing nearest neighbors in hitting times. This is our second contribution.

Finally, we develop a fully external-memory clustering algorithm (RWDISK) that uses only sequential sweeps over data files. This serves as a preprocessing step that yields the disk-resident clustered representation mentioned above, on top of which any nearest-neighbor algorithms can then be run.

We present extensive experimental results on real-world graphs with up-to 86 million edges. We show how tackling the high degree nodes boost both computational efficiency and link prediction accuracy; the improved performance of the deterministic algorithm over vanilla Monte Carlo simulations; and finally the finer quality of clusters returned by our clustering algorithm compared to a popular in-memory clustering algorithm METIS ([14]).

The paper is organized as follows: in section 3.1 we theoretically show the effect of high degree nodes on personalized pagerank (PPV). We also show the same for discounted hitting times by presenting a new result which expresses discounted hitting times in terms of PPV. In section 3.2 we present a deterministic local algorithm to compute top k nodes in personalized pagerank using these clusters. In section 3.5 we describe our RWDISK algorithm for clustering a disk resident graph by only using sequential sweeps of files. We conclude with experimental results on large disk-resident Live Journal, DBLP and Citeseer graphs.

2 Background and Related Work

In this section we will briefly describe interesting random walk based proximity measures, namely personalized pagerank and hitting times. We will also discuss the relevance of personalized pagerank for graph clustering.

PERSONALIZED PAGERANK. Consider a random walk starting at node a, such that at any step the walk can be reset to the start node with probability α. The stationary distribution corresponding to this stochastic process is defined as the personalized pagerank vector (PPV) of node a. The entry corresponding to node j in the PPV vector for node a is denoted by $PPV(a, j)$. Large values of $PPV(a, j)$ is indicative of higher similarity/relevance of node j w.r.t a. For a general restart probability distribution r personalized pagerank is defined as $v = \alpha r + (1 - \alpha)P^T v$. P is the row normalized probability transition matrix and $P^T v$ is the distribution after one step of random walk from v. Let's define x_t as the probability distribution over all nodes at timestep t. x_0 is defined as the probability distribution with 1.0 at the start node and zero elsewhere. By definition we have $x_t = P^T x_{t-1} = (P^T)^t x_0$. Let v_t be the partial sum of occupancy probabilities up-to timestep t. Now we can write PPV as:

$$v(j) = \sum_{t=1}^{\infty} \alpha(1 - \alpha)^{t-1} x_{t-1}(j) = \lim_{t \to \infty} v_t(j) \tag{1}$$

Personalized pagerank has been shown to have empirical benefits in keyword search [5], link prediction [15], fighting spam [13]; there has been an extensive literature on algorithms for computing them locally [3, 5], off-line [12, 9, 17], and from streaming data [18] etc.

DISCOUNTED HITTING TIME. Hitting time in random walks is a well-studied measure in probability theory [1]. Hitting times and other local variations of it has been used as a proximity measure for link prediction [15], recommender systems [4], query suggestion [16], manipulation resistant reputation systems [11] etc. We would use the following variation of hitting time. Note that this is closer to the original hitting time definition, and is different from the generalized hitting time defined in [10]. Consider a random walk which, once started from i stops if node j is encountered, or with probability α. The expected time to hit node j in this process is defined as the α discounted hitting time from node i to node j, ($h_\alpha(i, j)$). Similar to the undiscounted hitting time, this can be written as the average of the hitting times of its neighbors to j.

$$h_\alpha(i, j) = \begin{cases} 1 + (1 - \alpha) \sum_k P(i, k) h_\alpha(k, j) & \text{when } i \neq j \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The maximum value of this quantity is $1/\alpha$, which happens when node j is never hit.

GRAPH CLUSTERING. Recently there has been interesting theoretical work ([19, 2]) for using random walk based approaches for computing good quality local graph partitions (cluster) near a given anchor node. The main intuition is that a random walk started inside a low conductance cluster will mostly stay inside the cluster. Cluster-quality is measured by its conductance, which is defined as follows: For a subset of A of all nodes V, let $\Phi_V(A)$ denote conductance of A, and $\mu(A) = \sum_{i \in A} degree(i)$. As in [19], conductance is defined as:

$$\Phi_V(A) = \frac{E(A, V - A)}{min(\mu(A), \mu(V - A))} \quad (3)$$

A good-quality cluster has small conductance, resulting from a small number of cross-edges compared to the total number of edges. The smaller the conductance the better the cluster quality. Hence 0 is perfect score, for a disconnected partition, whereas 1 is the worst score for having a cluster with no intra-cluster edges. Conductance of a graph is defined as the minimum conductance of all subsets A.

The formal algorithm to compute a low conductance local partition near a pre-selected seed node was given in [19]. The idea is to compute sparse representation of probability distribution over the neighboring nodes of a seed node in order to return a local cluster with small conductance with high probability. The running time is nearly linear in the size of the cluster it outputs.

3   Proposed Work

There are two main problems with nearest neighbor computation in large real world networks. First, most local algorithms for computing nearest neighbors suffer from the presence of high degree nodes. In section 3.1 we propose a solution that converts high degree nodes to sinks. This effectively stops a random walk once it hits a high degree node, thus preventing a possible blow-up in the neighborhood-size in local algorithms. Our results imply that in power law graphs, this does not affect the proximity measures significantly.

The second issue is that of computing proximity measures on large disk-resident graphs. While there are existing external-memory algorithms for computing random walks in large disk-resident

3

graphs [9, 17], most of these store sketches aimed to compute one particular measure. Streaming algorithms [18] on the other hand require multiple passes over the entire data. While all these algorithms use interesting theoretical properties of random walks, they do not provide a generic framework for computing *arbitrary* random walk based proximity measures on the fly. One solution would be to cluster the graph and store each partition on a disk-page. Given such a clustered representation, one may easily simulate random walks, and thus compute nearest neighbors in hitting times, pagerank, simrank etc. Instead in section 3.2 we propose a deterministic local algorithm to compute nearest neighbors, which is later shown in section 4 to reduce number of page-faults compared to random simulations.

Finding a good clustering is a well-studied problem [2, 19]. Good clusters will have few cross edges, leading to self-contained random walks and less page-faults. Sometimes a good clustering can be achieved by using extra features, e.g. url's in the web-graph. However we are unaware of a fully external memory clustering algorithm in the general setting. Building on some ideas in prior literature [2], [17] we present an external memory clustering algorithm in section 3.3.

3.1 Effect of High Degree Nodes

Local algorithms estimate the similarity between the query node and others by examining local neighbor-hoods around the query node. These mostly rely on dynamic programming or power iteration like techniques which involve updating a node's value by combining that of its neighbors. As a result whenever a high degree node is encountered these algorithms have to examine a much larger neighborhood leading to performance bottlenecks. The authors in [17] use rounding in order to obtain sparse representations of personalized pagerank and simrank. However before rounding, the simrank/pagrank vectors can become very dense owing to the high degree nodes. For [3, 5] the authors maintain a priority queue to store the active neighborhood of the query node. Every time a high degree node is visited all its neighbors need to be enqueued, thus slowing both algorithms down. Because of the power-law degree distribution such high degree nodes often exist in real world networks. Although there are only a few of them, due to the small-world property these nodes are easily reachable from other nodes, and they are often encountered in random walks. We will discuss the effect of high degree nodes on two proximity measures, personalized pagerank and discounted hitting times. Our analysis of the effect of degree on hitting time, and personalized pagerank is for the case of undirected graphs, although the main theorems, i.e. 3.5 and 3.1 holds for any graph.

EFFECT ON PERSONALIZED PAGERANK. The main intuition behind this analysis is that a very high degree node passes on a small fraction of its value to the out-neighbors, which might not be significant enough to spend our computing resources on. We argue that stopping a random walk at a high degree node does not change the personalized pagerank value at other nodes which have relatively smaller degree. First we show that the error incurred in personalized pagerank is inversely proportional to the degree of the sink node. Next we analyze the error for introducing a set of sink nodes. We turn a high degree node into a sink by removing all the outgoing neighbors and adding one self-loop with probability one, to have a well-defined probability transition matrix P. We do not change any incoming edges.

We denote by $PPV(\boldsymbol{r}, j)$ the personalized pagerank value at node j w.r.t start distribution $\boldsymbol{r}$, and $PPV(i, j)$ denotes ppv value at node j w.r.t a random walk started at node i. Let $\widehat{PPV}$ be the

personalized pagerank w.r.t. start distribution r on the changed transition matrix.

Theorem 3.1. *In a graph G, if a node s is changed into a sink, then for any node $i \neq s$, the personalized pagerank in the new graph w.r.t start distribution $\boldsymbol{r}$ can be written as:*

$$\widehat{PPV}(\boldsymbol{r}, i) = PPV(\boldsymbol{r}, i) - PPV(s, i)\frac{PPV(\boldsymbol{r}, s)}{PPV(s, s)}$$

Given theorem 3.1 we will prove that if degree of s is much higher that than of i, then the error will be small. In order to do this we would need to examine the quantity $PPV(i, j)/PPV(j, j)$. Define the first occurrence probability $fa_\alpha(i, j)$. Consider a random walk which stops if it hits node j; if j is not hit, it stops with probability α. $fa_\alpha(i, j)$ is simply the probability of hitting a node j for the first time from node i, in this α-discounted walk. This is defined as:

$$fa_\alpha(i, j) = \begin{cases} (1 - \alpha) \sum_k P(i, k) fa_\alpha(j, k) & \text{when } i \neq j \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

Lemma 3.2. *The personalized pagerank from node i to node j can be expressed as:*

$$PPV(i, j) = fa_\alpha(i, j) \times PPV(j, j)$$

Sketch. As proven in [12, 9], the personalized pagerank from node i to node j is simply the probability that a length L path from i will end in j, where L is chosen from a geometric distribution with probability $P(L = t) = \alpha(1 - \alpha)^{t-1}$. Note that these paths can have multiple occurrences of j. If we condition on the first occurrence of j, then $PPV(i, j)$ would simply be probability of hitting j for the first time in an α discounted random walk times the personalized pagerank from j to itself. $\square$

Lemma 3.3. *The error introduced at node $i \neq s$ by converting node s into a sink can be upper bounded by $\frac{d_i}{d_s}$.*

Sketch. Note that by linearity of personalized pagerank vectors, we have $PPV(\boldsymbol{r}, i) = \sum_k \boldsymbol{r}_k PPV(k, i)$. Also as shown in the appendix, $PPV(s, i) \leq d_i PPV(i, s)/d_s \leq d_i/d_s$. Now, the above statement can be proved by combining linearity with $PPV(k, s)/PPV(s, s) = fa_\alpha(k, s) \leq 1$ (from lemma 3.2). $\square$

Hence if d_s is much larger than d_i then this error is small. Now we will present the error for converting a set of nodes S to a sink. The first step is to show that the error incurred by turning a number of high degree nodes into sinks is upper bounded by the sum of their individual errors. That can again be simplified to

Lemma 3.4. *If we convert all nodes in set $S = \{s_1, s_2, ..., s_k\}$ into sinks, then the error introduced at node $i \notin S$ can be upper bounded by $\dfrac{d_i}{\min_{s \in S} d_s}$.*

The proofs for theorem 3.1 and lemma 3.4 can be found in the appendix.

In real world networks the degree distribution often follows a power law, i.e. there are relatively fewer nodes with very large degree. And also most nodes have very low degree *relative* to these nodes. Hence we can make a few nodes into sinks and gain a lot of computational efficiency without losing much accuracy. We would like to mention that although the degree in the analysis is the weighted degree of a node, for experimental purposes we remove nodes with a large number of out-neighbors and not the weighted degree, since a node with a few high-weight edges can have a large weighted degree but it can pass on a considerable amount of probability to its neighbors.

EFFECT ON HITTING TIME. In order to see the effect of turning high degree nodes into sinks on discounted hitting times, we introduce the following result in this paper.

Theorem 3.5. *The α-discounted hitting time $h_\alpha(i,j)$ is related to personalized pagerank by:*

$$\boldsymbol{h}_\alpha(i,j) = \frac{1}{\alpha}\left[1 - \frac{PPV(i,j)}{PPV(j,j)}\right]$$

Sketch. First we show that $h_\alpha(i,j) = \frac{1}{\alpha}(1 - fa_\alpha(i,j))$. This can be easily verified by substituting this in the equation for hitting time, i.e. (2). This combined with lemma 3.2 gives the result. □

In this section we will only show the effect of deleting one high degree node on hitting time. The effect of removing a set of high degree nodes follows from the analysis of the last section and would be skipped for brevity. We denote by $\hat{h}_\alpha(i,j)$ the hitting time after we turn node s into a sink. By combining theorems 3.1 and 3.5 and algebraic manipulations, we get:

$$h_\alpha(i,j) - \frac{d_j}{\alpha^2 d_s} \leq \hat{h}_\alpha(i,j) \leq h_\alpha(i,j) + \frac{d_j}{\alpha^2 d_s} \tag{5}$$

We used the fact that $PPV(j,j) \geq \alpha$ and lemma 3.2 to obtain the above. This implies that turning a very high degree node into sink has a small effect on hitting time.

In this section we have given theoretical justification for changing the very-high degree nodes into sinks. We have shown its effects on two well known random walk based proximity measures. In the next two sections we would demonstrate algorithms to compute nearest neighbors on a clustered graph representation, and an external memory algorithm to compute clustering.

3.2 Nearest-neighbors on clustered graphs

Given a clustered representation, one can easily simulate random walks from a node, to obtain nearest neighbors in different proximity measures. While simulations are computationally cheap, they have a lot of variation, and for some real-world graphs they often lead to many page-faults, owing to the absence of well-defined clusters. In this section we discuss how to use the clusters for deterministic computation of nodes "close" to an arbitrary query. As the measure of "closeness" from i, we pick the degree-normalized personalized pagerank, i.e. $PPV(i,j)/d_j$. d_j is the weighted degree of node j. This is a truly personalized measure, in the sense that a popular

node gets a high score only if it has a very high personalized pagerank value. We will use the degree-normalized pagerank as a proximity measure for link prediction in our experiments as well.

We want to compute nearest neighbors in $PPV(i,j)/d_j$ from a node i. For an undirected graph, we have $PPV(i,j)/d_j = PPV(j,i)/d_i$. Hence it is equivalent to computing nearest neighbors in personalized pagerank to a node i. For an un-directed graph we can easily change these bounds to compute nearest neighbors in personalized pagerank *from* a node. For computing personalized pagerank *to a node*, we will make use of the dynamic programming technique introduced by [12] and further developed for computing sparse personalized pagerank vectors by [17]. For a given node i, the PPV from j to it, i.e. $PPV(j,i)$ can be written as

$$PPV^t(j,i) = \alpha\delta(i) + (1-\alpha) \sum_{k \in nbs(j)} PPV^{t-1}(k,i)$$

Now let us assume that j and i are in the same cluster S. Hence the same equation becomes

$$PPV^t(j,i) = \alpha\delta(i) + (1-\alpha) \left[\sum_{k \in nbs(j) \cap S} P(j,k)PPV^{t-1}(k,i) + \right.$$

$$\left. \sum_{k \in nbs(j) \cap \bar{S}} P(j,k)PPV^{t-1}(k,i) \right]$$

Since we do not have access to $PPV^{t-1}(k), k \notin S$, we will replace it with upper and lower bounds. The lower bound is simply zero, i.e. we pretend that S is completely disconnected to the rest of the graph. A random walk from outside S has to cross the boundary of S, $\delta(S)$ to hit node i. Hence $PPV(k,i) = \sum_{m \in \delta(S)} Pr_\alpha(X_m|X_{\delta(S)})PPV(m,i)$, where X_m denotes the event that *Random walk hits node m before any other boundary node for the first time*, and the event $X_{\delta(S)}$ denotes the event that the *random walk hits the boundary $\delta(S)$*. Since this is a convex sum over personalized pagerank values from the boundary nodes, this is upper bounded by $\max_{m \in \delta(S)} PPV(m,i)$. Hence we have the upper and lower bounds as follows:

$$lb^t(j,i) = \alpha\delta(i) + (1-\alpha) \sum_{k \in nbs(j) \cap S} lb^{t-1}(k,i)$$

$$ub^t(j,i) = \alpha\delta(i) + (1-\alpha) \left[\sum_{k \in nbs(j) \cap S} P(j,k)ub^{t-1}(k,i) + \right.$$

$$\left. \{1 - \sum_{k \in nbs(j) \cap S} P(j,k)\} \max_{m \in \delta(S)} ub^{t-1}(m,i) \right]$$

Since S is small in size, the power method suffices for computing these bounds, one could also use rounding methods introduced by [17]. At each iteration we maintain the upper and lower bounds for nodes within S, and at the global upper bound $\max_{m \in \delta(S)} ub^{t-1}(m,i)$. In order to expand S we bring in the clusters for x of the *external* neighbors of
$\arg\max_{m \in \delta(S)} ub^{t-1}(m,i)$. Once this *global upper bound* falls below a pre-specified small threshold γ, we use these bounds to compute approximate k closest neighbors in degree-normalized personalized pagerank.

The ranking step to obtain top k nodes using upper and lower bounds is simple: we return all nodes which have lower bound greater than the $k+1^{th}$ largest upper bound (when $k = 1$, kth largest is the largest probability). We denote this as ub_{k+1}. Since all nodes outside the cluster are guaranteed to have personalized pagerank smaller than the global upper bound, which in turn is smaller than γ, we know that the true $(k+1)^{th}$ largest probability will be smaller than ub_{k+1}. Hence any node with lower bound greater than ub_{k+1} is guaranteed to be greater than the $k+1^{th}$ largest probability. We use an additive slack, e.g. $(ub_{k+1} - \epsilon)$ in order to return the top k *approximately* large ppv nodes. The reason for using an additive slack is that, for larger values of ub_{k+1}, this behaves like a small relative error, whereas for small ub_{k+1} values it allows a large relative slack, which is useful since we do not want to spend energy on the tail of the rank list anyways. In our algorithm we initialize γ with 0.1 and keep decreasing it until the bounds are tight enough to return k largest nodes. Note that one could rank the probabilities using the lower bounds, and return top k of those after expanding the cluster a fixed number of times. This translates to a larger approximation slack.

What if we want to compute this on a graph with high degree nodes converted into sinks? Although this is not undirected anymore, using our error bounds from section 3.1 we can easily show that (skipped for brevity) if the difference between two personalized pagerank values $PPV(a, i)$ and $PPV(b, i)$ is larger than $\frac{d_i}{\min_{s_i \in S} d(s_i)}$ in the original graph, then a will have larger PPV value than b in the altered graph. Given that the networks follow a power law degree distribution, the minimum degree of the nodes made into sinks is considerably larger than d_i for most i, we see that the pairs which had a considerable gap in their original values should still have the same ordering. Note that for high degree nodes the ordering will have more error. However because of the expander like growth of the neighborhood of a high degree node, most nodes are far away from it leading to an uninteresting set of nearest neighbors anyways.

3.3 Clustered Representation on Disk

Now that we have discussed how to use a given clustered representation for computing nearest neighbors efficiently, we will present an algorithm to generate such a representation on disk. The intuition behind this representation is to use a set of anchor nodes and assign each remaining node to its "closest" anchor. Since personalized page-rank has been shown to yield good quality clusters [2], we use it as the measure of "closeness". Our algorithm starts with a random set of anchors, and compute personalized pagerank from them to the remaining nodes. Since all nodes might not be reached from this set of anchors, we iteratively add new anchors from the set of unreachable nodes, and the recompute the cluster assignments. Thus our clustering satisfies two properties: new anchors are far away from the existing anchors, and when the algorithm terminates, each node each node i is guaranteed to be assigned to its closest anchor. Even though the anchors are chosen randomly this should not affect the clustering significantly because, any node within a tight cluster can serve as the anchor for that cluster.

While clustering can be done based on personalized pagerank *from* or *to* a set of anchor nodes, one is not known to be better or worse than the other a priori. We use personalized pagerank *from the anchor nodes* as the measure of closeness. In [17] the authors presented a semi-external memory algorithm to compute personalized pagerank to a set of nodes. While the authors mention that their algorithm can be implemented purely via external memory manipulations, we close the loop

via external memory computation of personalized pagerank from a set of anchor nodes (algorithm RWDISK). While we also use the idea of rounding introduced by the authors, the analysis of approximation error is different from their analysis. We provide proof sketches for the main results in this section, details of the proof can be found in the appendix.

We will first present the basic idea behind RWDISK, and we will call it RWDISK-simple. We will demonstrate the algorithm on a simple line graph. Then we will show when the simple algorithm does not work, and present RWDISK.

3.4   Algorithm RWDISK-simple

Algorithm *RWDISK-simple* computes PPV values from a set of anchor nodes based solely on passes on input files. In the subsequent sections we will describe the full algorithm. First we will show how to compute x_t and v_t by doing simple join operations on files. We will demonstrate by computing PPV from node b in a line graph of 4 nodes (Figure 1). The RWDISK algorithm will sequentially read and write from four kinds of file. We will first introduce some notation. X_t denotes a random variable, which represents the node the random walk is visiting at time t. $P(X_0 = a)$ is the probability that the random walk started at node a.

- The *Edges* file remains constant and contains all the edges in the graph and the transition probabilities. Treating *Edges* as an ASCII file with one line per edge, each line is a triplet $\{sourcenode, destnode, p\}$, where `sourcenode` and `destnode` are strings representing nodes in the graph, and $p = P(X_t = \mathtt{destnode}|X_{t-1} = \mathtt{sourcenode})$. Number of lines in *Edges* equals number of edges in the graph. *Edges* is sorted lexicographically by sourcenode.

- At iteration t, the *Last* file contains the values for x_{t-1}. Thus each line in *Last* is $\{source, anchor, value\}$, where `value` equals $P(X_{t-1} = \mathtt{source}|X_0 = \mathtt{anchor})$. *Last* is sorted by source.

- At iteration t of the algorithm, the file *Newt* will contain the values for x_t, i.e. each line is $\{source, anchor, value\}$, where `value` equals $P(X_t = \mathtt{source}|X_0 = \mathtt{anchor})$. *Newt*, once construction is finished, will also be sorted by source. Needless to say, the *Newt* of iteration t becomes the *Last* of iteration $t + 1$.

- The final file is *Ans*. At iteration t of the algorithm, the file *Ans* represents the values for v_t. Thus each line in *Ans* is $\{source, anchor, value\}$, where $\mathtt{value} = \sum_{n=1}^{t} \alpha(1-\alpha)^{n-1} x_{n-1}(j)$. *Ans*, once construction is finished, will also be sorted by source.

We will now confirm that these files can be derived from each other by a series of merges and sorts. In step 1 of Algorithm RWDISK, using the simple 4-node graph in figure 1 with b as the sole anchor, we initialize *Last* as the single line $b, b, 1.0$, and *Ans* with the single line b, b, α.

In the next step we compute the *Newt* file from *Last* and *Edges* by a disk-based matrix multiplication, which is a simple join operation of two files. Both *Last* and *Newt* sum to 1, since these


Figure 1: Line Graph
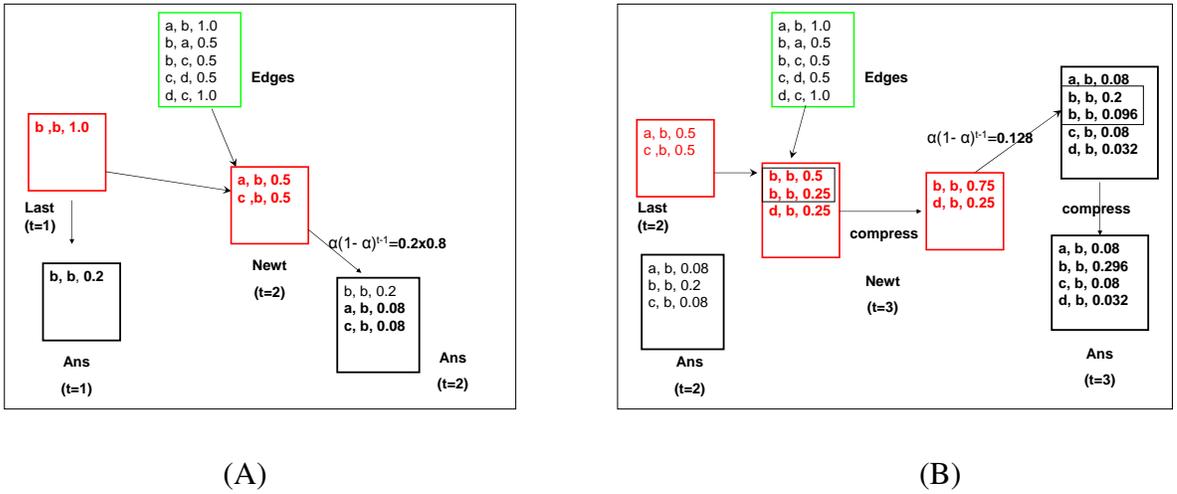
Figure 2: A. shows the first step of our algorithm on a line graph, and B. shows the second step. The input file (*Edges*) is in green, the intermediate files (*Last*, *Newt*) are in red and the output file (*Ans*) is in black.

are the occupancy probabilities at two consecutive timesteps. Once we have *Newt*, we multiply the probabilities by $\alpha(1-\alpha)^{t-1}$ (the probability that a random walk will stop at timestep t, if at any step the probability of stopping is α) and accumulate the values into the previous *Ans* file. Now the *Newt* file is renamed to *Last* file. Figure 2 is the same process on the new *Last*, i.e. old *Newt* file.

Now let us look at some details. For any node y it can appear many times in *Newt*, since the values in *Last* from different in-neighbors of y accumulate in *Newt*. For example in the *Newt* file at timestep 2, probability mass accumulates in b from both its incoming neighbors $\{c,d\}$. Hence we need to sort and compress *Newt*, in order to add all the different values. Compressing is the simple process where sets of consecutive lines in the sorted file that have the same source and anchor (and each have their own value) are replaced by one line containing the given source and anchor, and the sum of the values. Sorting and compression can each happen with O(n) sequential operations through the file (the former by means of bucket sort).

3.5 Algorithm RWDISK

The problem with the previous algorithm is that some of the intermediate files can become very large: much larger than the number of edges. Let N and E be the total number of nodes and edges in the graph. Let d be the average outdegree of a node. In most real-world networks within 4-5 steps it is possible to reach a huge fraction of the whole graph. Hence for any anchor the *Last* file might have N lines, if all nodes are reachable from that anchor. Therefore the *Last* file can have at most $O(AN)$ lines. The ultimate goal of the paper is to create a pagesize cluster (partition) for each anchor. If roughly n_a nodes can fit per page, then we would need N/n_a anchors. Hence the naive file join approach will lead to intermediate files of size $O(N^2/n_a)$. Since these files are also sorted every iteration, this will greatly affect both the runtime and disk-storage.

10

RWDISK(*Edges*, ϵ, α, **Anchors**)	
• Initialize: For each anchor $a \in$ *Anchors*,	
1. Append to each empty file *Last* the line a, 1.0.	
2. Append to each empty file *Ans* the line a, α.	**CreateNewtFile**(***Last***,*Edges*)
• $i = 1$.	• Read triplets $\{src,anchor,val\}$ from *Last*with common prefix "src"
• Create Newt files from a pass over the *Last* and *Edges* files. *Newt*← **CreateNewtFile**(*Last*,*Edges*).	• Read triplets $\{src,dst,prob\}$ from *Edges*with common prefix "src"
• If the *Newt* file is empty or $i >$ `maxiter`, return *Ans* after sorting and compressing it.	• Append to a temporary buffer Tmp the triplets $\{dst,anchor,prob \times val\}$
• Update the *Ans* file. *Ans*← **Update**(*Ans*,*Newt*,α,$i + 1$).	• If number of lines in $Tmp \geq M$ sort and compress Tmp, and merge with already sorted file *Newt*
• Round values in *Newt* files and copy to *Last* files. *Last*← **Round**(*Newt*,ϵ).	
• $\epsilon = \epsilon/\sqrt{1 - \alpha}$	
• Increment counter i. Goto step 3.	

Table 1: Pseudocode for RWDISK

Rounding for reducing file sizes: We address this serious problem by means of rounding tricks. In the step from *Newt* to *Last* at timestep t we round all values below ϵ_t to zero. Elements of a sparse rounded probability density vector sums to at most one. By rounding we only store the entries bigger than ϵ_{t-1}. Hence the total number of nonzero entries can be at most $1/\epsilon_{t-1}$. In fact, since the probability of stopping decreases by a factor of $(1 - \alpha)^t$ with t, we gradually increase ϵ_t, leading to sparser and sparser solutions.

As *Last* stores rounded x_{t-1} values for A anchors, its length can never exceed A/ϵ_{t-1}. Since *Newt* is obtained by spreading the probability mass along the out-neighbors of each node in *Last*, its total length can be roughly $A \times d/\epsilon_{t-1}$, where d is the average out-degree. For $A = N/n_a$ anchors, this value is $(N/n_a) \times d/\epsilon_{t-1} = E/(n_a \times \epsilon_{t-1})$. Without rounding this length was N^2/n_a.

In table 1 we present this rounding algorithm using only sequential scans of datafiles on disk. At one iteration of *CreateNewtFile*, for each outneighbor `dst` of node `src` appearing in *Last* file, we append a line $\{$`dst,anchor,prob` × `val`$\}$ to the *Newt* file. Since the length of *Last* is no more than A/ϵ_{t-1}, the total number of lines before sorting and compressing *Newt* can be $A \times d/\epsilon_{t-1}$. We avoid sorting a huge internal *Edges* every iteration by updating the *Newt* in a *lazy* fashion. We maintain a sorted *Newt* and append the triplets to a temporary buffer Tmp. Every time the number of lines exceed $M = 2A/\epsilon_{t-1}$, we sort the buffer, and merge it with the already sorted *Newt* file. Since the total number of lines in *Newt* can be as big as $A \times d/\epsilon_{t-1}$, this update happens roughly $d/2$ times, where d is the average out-degree.

The *update* function reads a triplet $\{$`source, anchor, val`$\}$ from *Newt* and appends it after

adjusting `val` by a factor of $\alpha(1-\alpha)^{iter-1}$ to *Ans*. Since *Ans* is not needed to generate the intermediate probability distributions, we do not sort and compress it in every iteration. We only do so once at the end. How big can the *Ans* file get? Since each *Newt* can be as big as $A \times d/\epsilon_{t-1}$, and we iterate for $maxiter$ times, the size of *Ans* can be roughly as big as $maxiter \times A \times d/\epsilon_{t-1}$. Since we increase ϵ_t every iteration, $\epsilon_t \geq \epsilon, \forall t$. Here are the file-sizes obtained from RWDISK in a nutshell:

Algorithm	# anchors	*Last* Size	*Newt* Size	*Ans* Size
No-rounding	$\frac{N}{n_a}$	$O(\frac{N^2}{n_a})$	$O(\frac{N^2}{n_a})$	$O(\frac{N^2}{n_a})$
Rounding	$\frac{N}{n_a}$	$O(\frac{N}{\epsilon \times n_a})$	$O(\frac{E}{\epsilon \times n_a})$	$O(\frac{\texttt{maxiter} \times E}{\epsilon \times n_a})$

3.6 Approximation Error

We present a proof sketch. The proofs of each lemma/theorem can be found in the appendix.

Error from rounding: The rounding step saves vast amounts of time and intermediate disk space, but how much error does it introduce? In this section we will describe the effect of rounding on computing the degree-normalized personalized pagerank values. First we will bound the error in personalized pagerank. $\Psi_\epsilon(x)$ is an operation from a vector to a vector which rounds all entries of x below ϵ to 0. Thus $\Psi_\epsilon(x[j])$ equals 0 if $x[j] \leq \epsilon$, and equals $x[j]$, otherwise.

We will denote $\hat{x}_t$ as the approximated x_t value from the rounding algorithm at iteration t. We want to have a bound on $v - \hat{v}$, where v is the exact personalized pagerank (equation (1)). In order to do that we will first bound $x_t - \hat{x}_t$, and then combine these errors to get the bound on the final error vector. We will denote $\bar{\epsilon}_t$ as a vector of errors accumulated at different nodes by rounding at time t. Note that $\bar{\epsilon}_t$ is strictly less than the probability vector x_t. Also note that $\hat{x}_t$ is strictly smaller than x_t. . Hence we can prove the following recursive equation.

Lemma 3.6. *We have $\hat{x}_t = \Psi_{\epsilon_t}(P^T \hat{x}_{t-1})$. Let $\bar{\epsilon}_t$ equal $P^T \hat{x}_{t-1} - \hat{x}_t$. At any node i we have $0 \leq \bar{\epsilon}_t(i) \leq x I(x \leq \epsilon_t) \leq \epsilon_t$, where $I(.)$ is the indicator function, and $x = P^T \hat{x}_{t-1}(i)$. Then we can prove $x_t - \hat{x}_t \leq \bar{\epsilon}_t + P^T(x_{t-1} - \hat{x}_{t-1})$.*

We will use this to bound the total error accumulated up-to time t in the probability vector x_t.

Lemma 3.7. *Let E_t denote $x_t - \hat{x}_t$, the vector of errors accumulated up to timestep t. E_t has only non-negative entries. We have $E_t \leq \sum_{r=1}^{t}(P^T)^{t-r} \bar{\epsilon}_r$.*

The above error equation shows how the epsilon-rounding accumulates over time. From equation (1) the total error incurred in the PPV value E can be bounded as follows.

Theorem 3.8. *Let E denote $v - \hat{v}$, and $PPV_\alpha(r)$ denote the personalized pagerank vector for start distribution r, and restart probability α. We have $E \leq \frac{1}{\alpha} PPV_\alpha(\bar{\epsilon})$ where $\bar{\epsilon}$ is a vector, with maximum entry smaller than $\frac{\alpha \epsilon}{1-\sqrt{1-\alpha}}$, and sum of entries less than 1.*

Theorem 3.8 indicates that our rounding scheme incurs an error in the personalized pagerank vector which is upper bounded by a constant times the stationary distribution of a α-restart walk whose start distribution is $\bar{\epsilon}$. For an undirected graph it can be shown (Appendix) that the total error at any node i can be at most $\dfrac{d(i)}{\delta}\dfrac{\epsilon}{1 - \sqrt{1 - \alpha}}$. where $d(i)$ is the weighted degree of node i, and δ is the minimum weighted degree. Since we are using *degree-normalized* personalized pagerank, the $d(i)$ at any node gets normalized leading to an error of $\dfrac{\epsilon}{\delta(1 - \sqrt{1 - \alpha})}$.

In fact this result turns out to be very similar to that of [2], which uses a similar idea to [3] for obtaining sparse representations of personalized pagerank vectors. The main difference is that RWDISK streams through the data without needing random access.

Error from early termination: Here is the error bound for terminating RWDISK early.

Theorem 3.9. *Let v_t be the partial sum of distributions up to time t. If we stop at $t = $ `maxiter`, then the error is given by $v - v_{maxiter} = (1 - \alpha)^{maxiter} PPV_\alpha(x_{maxiter})$. where $PPV_\alpha(x_{maxiter})$ is the personalized pagerank with start distribution $x_{maxiter}$.*

This theorem indicates that the total error incurred by early stopping is $(1 - \alpha)^{\texttt{maxiter}}$. Now we will combine theorems 3.8 and 3.9 to obtain the final approximation error guarantee.

Lemma 3.10. *If $\hat{v}_{maxiter}$ is obtained from RWDISK with parameters ϵ, α, `maxiter` and start distribution r then*

$$v - \hat{v}_{maxiter} \leq \frac{1}{\alpha} PPV_\alpha(\bar{\epsilon}) + (1 - \alpha)^{maxiter} PPV_\alpha(x_{maxiter})$$

where $x_{maxiter}$ is the probability distribution after `maxiter` steps, when the start distribution is r.

HIGH DEGREE NODES. In spite of rounding one problem with RWDISK is that if a node has high degree then it has large personalized pagerank value from many anchors and as a results can appear in a large number of $\{node, anchor\}$ pairs in the *Last* file. After the matrix multiplication each of these pairs now will lead to $\{nb, anchor\}$ pairs for each outgoing neighbor of the high degree node. Since we can only prune once the entire *Newt* file is computed the size can easily blow up. This is why RWDISK benefits from turning high degree nodes into sinks as described before in section 3.1.

THE CLUSTERING STEP. We randomly pick about 1 percent of the nodes as anchor points, and compute personalized pagerank from them by the RWDISK algorithm. Each node is assigned to the anchor node which has the largest pagerank value to it, and this assignment defines our graph clustering. However there might be orphan nodes after one pass of the algorithm: nodes which no anchor can reach. We need every node to be placed in exactly one cluster, and so if there are orphans, we go ahead and pick another set of random anchor nodes from the orphans from step 1, and compute personalized pagerank from them by rerunning RWDISK. For any batch of anchors we only store the information $\{src, closest\text{-}anchor, value\}$, where src is a node which is not an orphan. $closest\text{-}anchor$ is the anchor with the maximum PPV value among *all*

anchors seen so far, and `value` is the PPV value from that anchor to `src`. Now we reassign the nodes to the new anchors, in case some node found a closer anchor. We continue this process until there are no orphans left. The clustering satisfies two properties: 1) a new batch of anchors are far away from the existing pool of anchors, since we picked them from nodes which has PPV value 0 from the pre-existing pool of anchors; 2) after R rounds if the set of anchors is S_R, then each node i is guaranteed to be assigned to its closest anchor, i.e. $\arg\max_{a \in S_R} PPV(a, i)$.

4 Results

We present our results in three steps: first we show the effect of high degree nodes on i) computational complexity of RWDISK, ii) page-faults in random walk simulations for an actual link prediction experiment on the clustered representation, and iii) link prediction accuracy. Second, we show the effect of deterministic algorithms for nearest-neighbor computation on reducing the total number of page-faults by fetching the right clusters. Last, we compare the usefulness of the clusters obtained from RWDISK w.r.t a popular in-memory algorithm METIS.

DATA AND SYSTEM DETAILS. We present our results on three of the largest publicly available social and citation networks: a connected subgraph of the Citeseer co-authorship network, the entire DBLP corpus, and Live Journal (table 2). We used an undirected graph representation, although RWDISK can be used for directed graphs as well. The experiments were done on an off-the-shelf PC. We used a size 100 buffer and the *least recently used* replacement scheme. Each time a random walk moves to a cluster not already present in the buffer, the system incurs page-faults. We used a pagesize of $4KB$, which is standard in most computing environments.

Dataset	Size of *Edges*	Nodes	Edges	Median Degree
Citeseer	$24MB$	$100K$	$700K$	4
DBLP	$283MB$	$1.4M$	$12M$	5
LiveJournal	$1.4GB$	$4.8M$	$86M$	5

Table 2: # nodes,directed edges in graphs

4.1 Effect of High Degree Nodes

Turning high degree nodes into sinks have three-fold advantage: first, it drastically speeds up our external memory clustering; second, it reduces number of page-faults in random walk simulations done in order to rank nodes for link-prediction experiments; second it actually *improves* link-prediction accuracy.

EFFECT ON RWDISK. Table 3 contains running times of RWDISK on three graphs. For Citeseer, RWDISK algorithm completed roughly in an hour without introducing any sink nodes. For DBLP,

Dataset	Sink Nodes		Time
	Min degree	Number	
Citeseer	None	0	1.3 hours
DBLP	None	0	$\geq$ 2.5 days
	1000	900	**11** hours
LiveJournal	1000	950	60 hours
	100	$134,000$	**17** hours

Table 3: For each dataset, the minimum degree, above which nodes were turned into sinks, and the total number of sink nodes, time for RWDISK.

without degree-deletion, the experiments ran for above 2.5 days, after which they were stopped. After turning nodes with degree higher than 1000, the time was reduced to 11 hours, a larger than 5.5 fold speedup. The Live-Journal graph is the largest and most dense of all three. After we made nodes of degree higher than 1000 into sinks, the algorithm took 60 hours, which was reduced to 17 ($\geq$ 3 fold speedup) after removing nodes above degree 100. In table 2 note that, for both DBLP and LiveJournal the median degree is much smaller than the minimum degree of nodes converted into sink nodes. This combined with our analysis in section 3.1 confirms that we did achieve a huge computational gain without sacrificing the quality of approximation.

LINK PREDICTION. For link-prediction we used degree-normalized personalized pagerank as the proximity measure for predicting missing links. We picked the same set of 1000 nodes and the same set of links from each graph before and after turning the high degree nodes into sinks. For each node i we held out $1/3^{rd}$ of its edges and reported the percentage of held-out neighbors in top 10 ranked nodes in degree-normalized personalized pagerank from i. Only nodes below degree 100 and above degree 3 were candidates for link deletion, so that no sink node can ever be a candidate. From each node 50 random walks of length 20 were executed. Note that this is not AUC score; so a random prediction does much worse that 0.5 in these tasks.

From table 4 we see that turning high-degree nodes into sinks not only decrease page-faults by a factor of ~ 7, it also boosts the link prediction accuracy by a factor of 4 on average. Here is the

Dataset	Sink nodes	Accuracy	Page-faults
LiveJournal	none	0.2	1502
	degree above 100	**0.43**	**255**
DBLP	none	0.1	1881
	degree above 1000	**0.58**	**231**
Citeseer	none	0.79	69
	degree above 100	0.74	67

Table 4: Mean link-pred. acc. and pagefaults

reason behind the surprising trend in link prediction scores. The fact that page-faults decrease after introducing sink nodes is obvious, since in the original graph every time a node hits a high degree node there is higher chance of incurring page-faults. We believe that the link prediction accuracy is related to quality of clusters, and transitivity of relationships in a graph. More specifically in a *well-knit* cluster, two connected nodes do not just share one edge, they are also connected by many short paths, which makes link-prediction easy. On the other hand if a graph has a more expander-like structure, then in random-walk based proximity measures, everyone ends up being far away from everyone else. This leads to poor link prediction scores. In table 4 one can catch the trend of link prediction scores from worse to better from LiveJournal to Citeseer. Our intuition about the relationship between cluster quality and predictability is reflected in figure 3, where we see that LiveJournal has worse page-fault/conductance scores than DBLP, which in turn has worse scores than Citeseer. Within each dataset, we see that turning high degree nodes into a sink generally helps link prediction, which is probably because it also improves the cluster-quality. Are all high-degree nodes harmful? In DBLP high degree nodes without exception end up being words which can confuse random walks. However the Citeseer graph only contains author-author connections, and hence has relevant high degree nodes, which is probably why the link-prediction accuracy decreases slightly when we introduce sink nodes.

4.2 Deterministic vs. Simulations

We present the mean and median number of pagefaults incurred by the deterministic algorithm in section 3.2. We executed the algorithm for computing top 10 neighbors with approximation slack 0.005 for 500 randomly picked nodes. For Citeseer we computed the nearest neighbors in

Dataset	Mean Page-faults	Median Page-faults
LiveJournal	64	29
DBLP	54	16.5
Citeseer	6	2

Table 5: Page-faults for computing 10 nearest neighbors using lower and upper bounds

the original graph, whereas for DBLP we turned nodes with degree above 1000 into sinks and for LiveJournal we turned nodes with degree above 100 into sinks. Both mean and median pagefaults decrease from LiveJournal to Citeseer, showing the increasing cluster-quality, as is evident from the previous results. The difference between mean and median reveals that for some nodes the neighborhood is explored much more in order to compute the top 10 nodes. Upon closer investigation we found that for high degree nodes, the clusters have a lot of boundary nodes and hence the bounds are hard to tighten. Also from high degree nodes all other nodes are more or less farther away. In contrast to random simulations (table 4), these results show the superiority of the deterministic algorithm over random simulations in terms of number of page-faults (roughly 5 fold improvement).

4.3 RWDISK vs. METIS

We used `maxiter` $= 30$, $\alpha = 0.1$ and $\epsilon = 0.001$ for PPV computation. We use PPV and RWDISK interchangeably in this section. Note that $\alpha = 0.1$ in our random-walk setting is equivalent to a restart probability of $\alpha/(2 - \alpha) = 0.05$ in the lazy random walk setting of [2].

We used METIS as a baseline algorithm [14][1], which is a state of the art *in memory* graph partitioning algorithm. We used METIS to break dblp into about $50,000$ parts, which used **20 GB** of RAM, and LiveJournal into about $75,000$ parts which used **50 GB** of RAM. Since METIS was creating comparably larger clusters we tried to divide the Live Journal graph into $100,000$ parts, however the memory requirement was **80 GB** which was prohibitively large for us. In comparison RWDISK can be executed on a $2 - 4$ GB standard computing unit. Table 2 contains the details of the three different graphs and table 3 contains running times of RWDISK on these. Although the clusters are computed after turning high degree nodes into sinks, the comparison with METIS is done on the original graphs.

MEASURE OF CLUSTER QUALITY. A good disk-based clustering must combine two character-
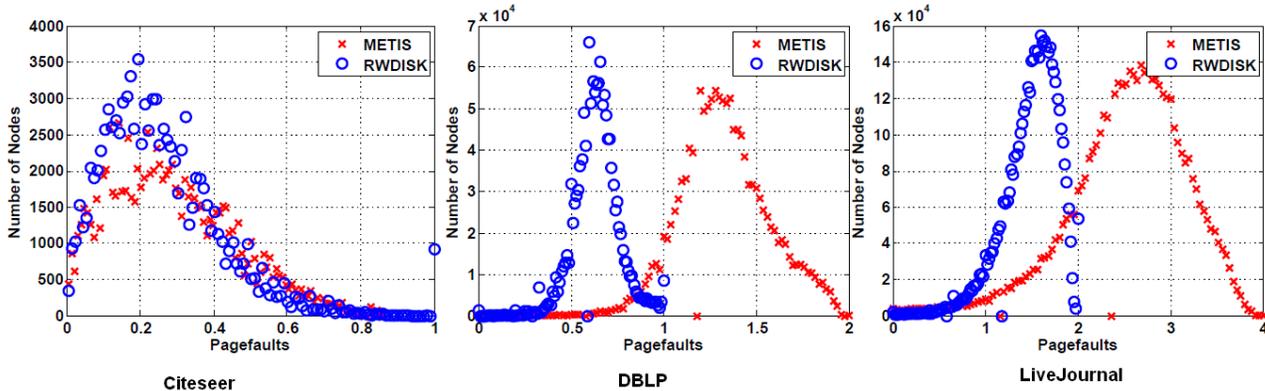
Figure 3: The histograms for the expected number of pagefaults if a random walk stepped outside a cluster for a randomly picked node. Left to right the panels are for Citeseer, DBLP and LiveJournal.

istics: (a) the clusters should have low conductance, and (b) they should fit in disk-sized pages. Now, the graph conductance ϕ measures the average number of times a random walk can escape outside a cluster [19], and each such escape requires the loading of one new cluster, causing an average of m page-faults ($m = 1$ if each cluster fits inside one page). Thus, $\phi \cdot m$ is the average number of page-faults incurred by one step of a random walk; we use this as our overall measure of cluster quality. Note that m here is the expected size (in pages) of the cluster that a randomly picked node belongs to, and this is *not* necessarily the average number of pages per cluster.

Briefly, figure 3 tells us that in a single step random walk METIS will lead to similar number of pagefaults on Citeseer, about $1/2$ pagefaults more than RWDISK on DBLP and 1 more in *LiveJournal*. Hence in a 20 step random walk METIS will lead to about 5 more pagefaults than RWDISK on DBLP and 20 more pagefaults on LiveJournal. Note that since a new cluster can be much larger than a disk-page size its possible to make more than 20 pagefaults on a 20 step random

---

[1]The software for partitioning power law graphs has not yet been released.

walk in our paradigm. In order to demonstrate the accuracy of this measure we actually simulated 50 random walks of length 20 from 100 randomly picked nodes from the three different graphs. We noted the average page-faults and average time in wall-clock seconds. Figure 4 shows how many *more pagefaults METIS incurs than RWDISK in every simulation*. The wallclock seconds is the **total time** taken for all 50 simulations averaged over the 100 random nodes. These numbers exactly match our expectation from figure 3. We see that on Citeseer METIS and RWDISK gives
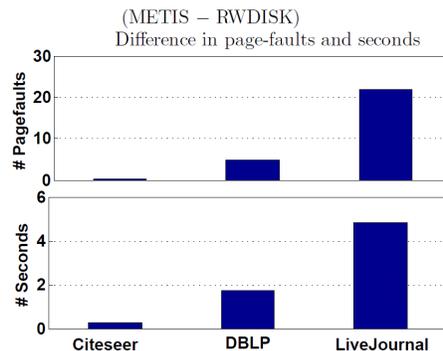
Figure 4: #Page-faults(METIS)-#Page-faults(RWDISK) per 20 step random walk in upper panel. Bottom Panel contains total time for simulating 50 such random walks. Both are averaged over 100 randomly picked source nodes.

comparable cluster qualities, but on DBLP and LiveJournal RWDISK performs much better.

5   Conclusion

This paper address the following problem. Random-walk based measures of proximity in graphs, such as Personalized Page Rank, Hitting Times and Commute times are becoming very important and popular, and yet there are limitations to what we can do when graphs become enormous. This paper introduces analysis and algorithms which try to address this in a generalizable way: not specific to one kind of graph partitioning nor one specific proximity measure. We take two steps. First, we identify the serious role played by high degree nodes in damaging computational complexity, and we prove that a simple transform of the graph can mitigate the damage with bounded impact on accuracy. Second, we apply the result to produce algorithms for the two components of general-purpose proximity queries on enormous graphs: algorithms to rank top-n neighbors by a broad class of random-walk based proximity measures including PPV, and a graph partitioning step to distribute graphs over a file system or over nodes of a distributed compute-node cluster. In future work we will experiment with a highly optimized implementation designed to respect true disk page size and hope to give results on graphs with billions of edges.

References

[1] David Aldous and James Allen Fill. *Reversible Markov Chains*. 2001.

[2] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *FOCS*, 2006.

[3] P. Berkhin. Bookmark-Coloring Algorithm for Personalized PageRank Computing. *Internet Mathematics*, 2006.

[4] M. Brand. A Random Walks Perspective on Maximizing Satisfaction and Profit. In *SIAM '05*, 2005.

[5] Soumen Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW '07*, New York, NY, USA.

[6] Soumen Chakrabarti, Jeetendra Mirchandani, and Arnab Nandi. Spin: searching personal information networks. In *SIGIR '05*.

[7] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD '09*.

[8] Bhavana Bharat Dalvi, Meghana Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *Proc. VLDB Endow.*, 1(1):1189–1204, 2008.

[9] D. Fogaras, B. Rcz, K. Csalogny, and Tams Sarls. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2004.

[10] Fan Chung Graham and Wenbo Zhao. Pagerank and random walks on graphs. In *"Fete of Combinatorics"*.

[11] John Hopcroft and Daniel Sheldon. Manipulation-resistant reputations using hitting time. Technical report, Cornell University, 2007.

[12] G. Jeh and J. Widom. Scaling personalized web search. In *Stanford University Technical Report*, 2002.

[13] Amruta Joshi, Ravi Kumar, Benjamin Reed, and Andrew Tomkins. Anchor-based proximity measures. In *WWW '07*.

[14] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[15] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *CIKM '03*, 2003.

[16] Qiaozhu Mei, Dengyong Zhou, and Kenneth Church. Query suggestion using hitting time. In *CIKM '08*.

[17] Tamás Sarlós, Adrás A. Benczúr, Károly Csalogány, Dániel Fogaras, and Balázs Rácz. To randomize or not to randomize: space optimal summaries for hyperlink analysis. In *WWW*, 2006.

[18] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. In *PODS*, 2008.

[19] D. Spielman and S. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the STOC'04*.

[20] Yen yu Chen, Qingqing Gan, and Torsten Suel. Local methods for estimating pagerank values. In *In CIKM*, pages 381–389. ACM Press, 2004.

6   Appendix

SYMMETRY OF DEGREE NORMALIZED PPV IN UNDIRECTED GRAPHS. This follows directly from the reversibility of random walks.

$$v_i(j) \;=\; \alpha \sum_{t=0}^{\infty}(1-\alpha)^t P^t(i,j) = d_j/d_i\, \alpha \sum_{t=0}^{\infty}(1-\alpha)^t P^t(j,i)$$
$$\Rightarrow v_i(j)/d_j \;=\; v_j(i)/d_i \tag{6}$$

PROOF OF LEMMA 3.6. We have $\hat{x}_t = \Psi_{\epsilon_t}(P^T \hat{x}_{t-1})$. Also $\bar{\epsilon}_t$ denotes the difference $P^T\hat{x}_{t-1} - \hat{x}_t$. Let $y = P^T\hat{x}_{t-1}(i)$. Note that $\bar{\epsilon}_t(i)$ is 0 if $y \geq \epsilon_t$, and is y if $y < \epsilon_t$. Since $\bar{\epsilon}_t(i) \leq P^T\hat{x}_{t-1}(i)$, for all t ϵ_t has the property that its sum is at most 1, and the maximum element is at most ϵ_t. This gives:

$$\mathbf{x_t} - \hat{\mathbf{x}}_{\mathbf{t}} = x_t - \Psi_{\epsilon_t}(P^T\hat{x}_{t-1}) = x_t - (P^T\hat{x}_{t-1} - \bar{\epsilon}_t)$$
$$\leq \bar{\epsilon}_{\mathbf{t}} + \mathbf{P^T}(\mathbf{x_{t-1}} - \hat{\mathbf{x}}_{\mathbf{t-1}})$$

PROOF OF LEMMA 3.7. Let's denote $E_t = x_t - \hat{x}_t$ by the vector of errors accumulated *up to timestep* t. Note that this is always going to have non-negative entries.

$$\mathbf{E_t} \leq \bar{\epsilon}_t + P^T E_{t-1} \leq \bar{\epsilon}_t + P^T \bar{\epsilon}_{t-1} + (P^T)^2 \bar{\epsilon}_{t-2} + ...$$
$$\leq \sum_{\mathbf{r=1}}^{\mathbf{t}}(\mathbf{P^T})^{\mathbf{t-r}}\bar{\epsilon}_{\mathbf{r}}$$

PROOF OF THEOREM 3.8. By plugging in the result from lemma 3.7 into equation (1) the total error incurred in the PPV value is

$$\mathbf{E} = |v - \hat{v}| \leq \sum_{t=1}^{\infty}\alpha(1-\alpha)^{t-1}(x_t - \hat{x}_t)$$
$$\leq \sum_{t=1}^{\infty}\alpha(1-\alpha)^{t-1}E_t \leq \sum_{t=1}^{\infty}\alpha(1-\alpha)^{t-1}\sum_{r=1}^{t}(P^T)^{t-r}\bar{\epsilon}_r$$
$$\leq \sum_{r=1}^{\infty}\sum_{t=r}^{\infty}\alpha(1-\alpha)^{t-1}(P^T)^{t-r}\bar{\epsilon}_r$$
$$\leq \sum_{r=1}^{\infty}(1-\alpha)^{r-1}\sum_{t=r}^{\infty}\alpha(1-\alpha)^{t-r}(P^T)^{t-r}\bar{\epsilon}_r$$
$$\leq \sum_{r=1}^{\infty}(1-\alpha)^{r-1}\left[\sum_{t=0}^{\infty}\alpha(1-\alpha)^{t}(P^T)^{t}\right]\bar{\epsilon}_r$$
$$\leq \frac{1}{\alpha}\left[\sum_{\mathbf{t=0}}^{\infty}\alpha(\mathbf{1-\alpha})^{\mathbf{t}}(\mathbf{P^T})^{\mathbf{t}}\right]\sum_{\mathbf{r=1}}^{\infty}\alpha(\mathbf{1-\alpha})^{\mathbf{r-1}}\bar{\epsilon}_{\mathbf{r}}$$

We used $\epsilon_r = \epsilon_{r-1}/\sqrt{1-\alpha} = \epsilon/(\sqrt{\alpha-1})^{r-1}$. Note that $\bar{\epsilon}_r$ is a vector such that $|\bar{\epsilon}_r|_\infty \leq \epsilon_r \leq \epsilon/(\sqrt{1-\alpha})^{r-1}$, and $|\bar{\epsilon}_r|_1 \leq 1$. Let $\bar{\epsilon}$ equal vector $\sum_{r=1}^{\infty} \alpha(1-\alpha)^{r-1}\bar{\epsilon}_r$. Clearly entries of $\bar{\epsilon}$ sum to at most 1, and the largest entry can be at most $\alpha \sum_{r=1}^{\infty}(1-\alpha)^{r-1}\frac{\epsilon}{\sqrt{1-\alpha}^{r-1}} = \frac{\epsilon\alpha}{1-\sqrt{1-\alpha}}$.

Now we have $E \leq \frac{1}{\alpha}PPV_\alpha(\bar{\epsilon})$. The above is true because, by definition $\left[\sum_{t=0}^{\infty}\alpha(1-\alpha)^t(P^T)^t\right]\bar{\epsilon}$ equals the personalized pagerank with start distribution $\bar{\epsilon}$, and restart probability α. We will analyze this for an undirected graph.

$$
\begin{aligned}
PPV_\alpha(\bar{\epsilon}, i) &= \sum_j \sum_{t=0}^{\infty}\alpha(1-\alpha)^t(P^T)^t(i,j)\bar{\epsilon}(j) \\
&\leq \max_j \bar{\epsilon}(j) \sum_{t=0}^{\infty}\alpha(1-\alpha)^t \sum_j P^t(j,i) \\
&= \max_j \bar{\epsilon}(j) \sum_{t=0}^{\infty}\alpha(1-\alpha)^t \sum_j \frac{d_i P^t(i,j)}{d_j} \\
&\leq \frac{d_i}{\delta}\max_j \bar{\epsilon}(j) \sum_{t=0}^{\infty}\alpha(1-\alpha)^t \sum_j P^t(i,j) \\
&\leq \frac{d_i}{\delta}\max_j \bar{\epsilon}(j) \sum_{t=0}^{\infty}\alpha(1-\alpha)^t \\
&\leq \frac{d_i}{\delta}\max_j \bar{\epsilon}(j)
\end{aligned}
$$

The fourth step uses the reversibility of random walks for undirected graphs, i.e. $d_i P^t(i,j) = d_j P^t(j,i)$, where d_i is the weighted degree of node i. δ is the minimum weighted degree. Thus we have $E(i) \leq \frac{d_i}{\delta}\frac{\epsilon}{1-\sqrt{1-\alpha}}$.

PROOF OF THEOREM 3.9. Let r be the start distribution, and v_t be the partial sum upto time t. We know that $\forall t \geq \texttt{maxiter}, x_t = (P^T)^{t-\texttt{maxiter}}x_{\texttt{maxiter}}$

$$
\begin{aligned}
v - v_{\texttt{maxiter}} &= \sum_{t=\texttt{maxiter}+1}^{\infty}\alpha(1-\alpha)^{t-1}x_{t-1} \\
&= (1-\alpha)^{\texttt{maxiter}}PPV_\alpha(x_{\texttt{maxiter}})
\end{aligned}
$$

PROOF OF LEMMA 3.10. Let $\hat{v}$ be rounded PPV values when $\texttt{maxiter}$ equals ∞, and $\hat{v}_{\texttt{maxiter}}$ the rounded values from RWDISK. We have

$$
v - \hat{v}_{\texttt{maxiter}} = (v - \hat{v}) + (\hat{v} - \hat{v}_{\texttt{maxiter}})
$$

Using the same idea as theorem 3.9 and the fact that $\hat{x}_t(i) \leq x_t(i), \forall i$, we can upper bound the difference $\hat{v} - \hat{v}_{\texttt{maxiter}}$ by $(1-\alpha)^{\texttt{maxiter}}PPV_\alpha(x_{\texttt{maxiter}})$. This, combined with theorem 3.8 gives the desired result.

PROOF OF THEOREM 3.1. Personalized pagerank of a start distribution r can be written as

$$
PPV(r) = \alpha r + \alpha \sum_{t=1}^{\infty}(1-\alpha)^t(P^T)^t r = \alpha r + (1-\alpha)PPV(P^T r) \tag{7}
$$

By turning node s into a sink, we are *only* changing the sth row of P. We denote by r_s the indicator vector for node s. For $r = r_s$ we have personalized pagerank from node s. Essentially we are subtracting the entire row $P(s,:) = P^T r_s$ and adding back r_s. This is equivalent to subtracting

21

the matrix $\boldsymbol{vu}^T$ from P, where $\boldsymbol{v}$ is $\boldsymbol{r_s}$ and $\boldsymbol{u}$ is defined as $P^T\boldsymbol{r_s} - \boldsymbol{r_s}$. $PPV(\boldsymbol{r}) = \alpha(I - (1 - \alpha)P^T + (1-\alpha)\boldsymbol{uv}^T)^{-1}\boldsymbol{r}$ Let $M = I - (1-\alpha)P^T$. Hence $PPV(\boldsymbol{r}) = \alpha M^{-1}\boldsymbol{r}$. A straightforward application of the Sherman Morrison lemma gives $\widehat{PPV}(\boldsymbol{r}) = PPV(\boldsymbol{r}) - \alpha(1-\alpha)\frac{M^{-1}\boldsymbol{uv}^T M^{-1}}{1+(1-\alpha)\boldsymbol{v}^T M^{-1}\boldsymbol{u}}\boldsymbol{r}$ Note that $M^{-1}\boldsymbol{u}$ is simply $1/\alpha[PPV(P^T\boldsymbol{r_s}) - PPV(\boldsymbol{r_s})]$ and $M^{-1}\boldsymbol{r}$ is simply $1/\alpha PPV(\boldsymbol{r})$. Also $\boldsymbol{v}^T PPV(\boldsymbol{r})$ equals $PPV(\boldsymbol{r}, s)$. Combining these facts with equation 7 yields the following:

$$\widehat{PPV}(\boldsymbol{r}) = PPV(\boldsymbol{r}) - [PPV(\boldsymbol{r_s}) - \boldsymbol{r_s}]\frac{PPV(\boldsymbol{r}, s)}{PPV(\boldsymbol{r_s}, s)}$$

This leads to the element-wise error bound in theorem 3.1.

PROOF OF LEMMA 3.4. For proving the above we use a series of sink node operations on a graph and upper bound each term in the sum. For $j \leq k$ $S[j]$ denote the subset $\{s_1, ..., s_j\}$ of S. Also let $G \setminus S[j]$ denote a graph where we have made each of the nodes in $S[j]$ a sink. $S[0]$ is the empty set and $G \setminus S[0] = G$. Since we do *not* change the outgoing neighbors of any node when make a node into a sink, we have $G \setminus S[j] = (G \setminus S[j-1]) \setminus s_j$. Which leads to:

$$PPV^{G \setminus S[k-1]}(\boldsymbol{r}, i) - PPV^{G \setminus S[k]}(\boldsymbol{r}, i)$$
$$\leq \frac{PPV^{G \setminus S[k-1]}(s_k, i)PPV^{G \setminus S[k-1]}(\boldsymbol{r}, s_k)}{PPV(s_k, s_k)}$$
$$\leq PPV^{G \setminus S[k-1]}(s_k, i)$$

The last step can be obtained by combining linearity of personalized pagerank with lemma 3.2. Now, using a telescoping sum:

$$PPV^G(\boldsymbol{r}, i) - PPV^{G \setminus S[k]}(\boldsymbol{r}, i) \leq \sum_{j=1}^{k} PPV^{G \setminus S[j-1]}(s_j, i)$$

The above equation also shows that by making a number of nodes sink the personalized pagerank value w.r.t any start distribution at a node can only decrease, which intuitively makes sense. Thus each term $PPV^{G \setminus S[k-1]}(s_k, i)$ can be upper bounded by $PPV^G(s_k, i)$, and $PPV^{G \setminus S[k-1]}(\boldsymbol{r}, s_k)$ by $PPV^G(\boldsymbol{r}, s_k)$. This gives us the following sum, which can be simplified using (6) as,

$$\sum_{j=1}^{k} PPV(s_j, i) = \sum_{j=1}^{k} \frac{d_i PPV(i, s_j)}{d(s_j)} \leq \frac{d_i}{\min_{s \in S} d_s}$$

The last step follows from the fact that $PPV(i, s_j)$, summed over j has to be smaller than one. This leads to the final result in lemma 3.4.

ML

MACHINE LEARNING
D E P A R T M E N T

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Carnegie Mellon.