A Scalable External Memory Graph Partitioning Algorithm to Reduce Disk-seeks in Random Walks

Purnamrita Sarkar
Machine Learning Department
Carnegie Mellon University
psarkar@cs.cmu.edu

Andrew W. Moore
Google Inc.
Pittsburgh, PA
awm@google.com

ABSTRACT

This paper introduces an algorithm to generate efficient disk-resident representations of graphs that are too large to fit in main memory. The representation is efficient in the sense that stepping through paths of connected nodes in the graph (e.g. simulating a random walk) involves relatively few disk seeks. Since a broad class of graph-based similarity metrics (e.g. personalized pagerank, hitting and commute times, simrank) can be estimated by simulating random walks in a graph, this algorithm will strongly benefit applications such as link prediction in social networks, personalized graph search, fraud detection and collaborative filtering. We build on previous work that has shown that personalized pagerank based graph-partitioning has desirable properties, and our primary contribution is an efficient algorithm based purely on sequential sweeps through files to generate such partitions. In addition to the disk-based algorithm, this paper provides error bounds for computing approximate personalized pagerank vectors, which can be used to find good local partitions (graph cuts with small conductance) near a given starting point in a graph. We also present some new theoretical properties of personalized pagerank which enables us to make our algorithm tractable on real world graphs with up-to 69 million edges. We empirically demonstrate the utility of the disk-based partitioning for link prediction tasks on real world disk-resident graphs like DBLP, Citeseer and Live-Journal. In addition to vanilla random walk simulations, we present deterministic algorithms for quickly computing top k neighbors of a node in personalized pagerank using our disk-resident graph representation.

1. INTRODUCTION

A number of important graph-based real world applications, (e.g. collaborative filtering in recommender networks, link prediction in social networks, fraud detection, and personalized graph search techniques) rely on graph-theoretic measures of similarity. Random walk based proximity measures have been widely used for these applications. Most of these proximity measures, e.g. personalized pagerank [8], truncated hitting and commute times [11], simrank [9] can be accurately estimated by simulating short random walks in a graph. Typical queries used in Personalized Page Rank vectors (PPV) can be: "which are the k most probable nodes to visit if I start randomly at one of these four query nodes?" Existing algorithms for query processing on massive graphs often assume that the graph can be loaded into main memory. What if one needs to process an *arbitrary* query on a graph too large to fit into main memory? For a disk-resident graph, a straightforward implementation of a random walk will give rise to too many disk seeks, since it relies on random access to the outgoing neighbors of the current node.

The out-of-memory constraint is realistic since many graphs derived from social networks, web structure and intelligence analysis are too big. 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, the graph could be too large, e.g. the Web graph. 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 [5], 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 [6].

There are two constraints for this setting: (a) the space of queries is too large for us to precompute and cache answers for all possible queries and (b) the entire graph cannot be loaded into main memory. Existing algorithms relax one or both of these constraints. For example streaming algorithms [13] can process graphs too big for memory, but whenever a new query arrives they can require a few passes of the data. Algorithms like [7] use external memory algorithms for computing and storing random walk fingerprints from all nodes by using simple join operations on files. However for simulating a random walk from an arbitrary query one would need to make many sequential passes through the file of edges in order to obtain out-neighbors of a current node, which will incur too much overhead at query time.

In this paper we present and demonstrate an algorithm that uses only sequential scans for partitioning a disk-resident graph, and storing each partition in its own disk page with relative few cross-page links. Using this we can simulate arbitrary random walks by loading the appropriate pages into memory. Similar to previous work in [1] we use degree-normalized personalized pagerank as the proximity measure for our disk-base partitioning. Our other contribution is that we theoretically and empirically show that high degree nodes do not contribute very much to the personalized pagerank values, rather they sometimes harm tasks like link prediction, by distracting random walks. This observation

WWW2010, April 26-30, 2010, Raleigh, North Carolina.
.

enables us to make our algorithm more efficient, and scale it to up-to networks with about 86 million directed edges.

The rest of the paper is organized as follows: in section 2 we describe our RWDISK algorithm for computing approximate personalized pagerank only using sequential sweeps of files, and provide theoretical error bounds for the approximation. In section 4 we show the effect of high degree nodes on personalized pagerank, and use this for making our algorithm computationally more efficient. In section 5 we present a deterministic algorithm to compute top k nodes in degree-normalized personalized pagerank using these clusters. We conclude with experimental results on large disk-resident Live Journal, DBLP and Citeseer graphs.

2. COMPUTING PERSONALIZED PAGERANK VIA ROUNDING

The key to our algorithm is to use a large number of "anchor" nodes as seeds and assign the remaining nodes to the anchor closest to it. We will explain and demonstrate the desirability of this approach towards obtaining good partitions. We use personalized pagerank from an anchor as a measure of "closeness". First, we will define personalized pagerank, then present a simple file-join based algorithm *RWDISK-simple* which uses only sequential scans of the files. We will show why the simple approach does not scale to large graphs, and present the improved *RWDISK* algorithm. We will conclude by presenting the approximation error incurred by RWDISK.

2.1 Summary of personalized page rank

Since Personalized Pagerank vectors (PPV values) were first introduced by [8], an important literature has grown up around their theoretical properties (e.g [1]) and their empirical benefits for information retrieval in graphs ([2]). They are defined by the simple concept of a random walk with probability α of resetting to the start node. 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 $v_a(j)$. If $v_a(j)$ is relatively large then node j is likely to be relatively more relevant to someone who is interested in node a. Whereas pagerank ([4]) gives a measure of global importance, personalized pagerank gives a measure of importance of any node w.r.t the start node. For a general restart probability distribution r personalized pagerank is defined as $v = \alpha r + (1-\alpha)P^T v$.

$P^T v$ is the distribution after one step of random walk from v. Let $v_i(j)$ denote the personalized pagerank of node j w.r.t the starting node i. We will represent PPV as the weighted sum of occupancy probabilities at timesteps $t = 0$ to $t = \infty$, i.e. $\sum_{t=1}^{\infty} \alpha(1-\alpha)^{t-1}P^{t-1}(i,j)$. This definition can also be found in[8]. For simplicity we will drop the subscript and fix the starting node at i, unless otherwise mentioned. Let's define x_t as the probability distribution over all nodes at timestep t, when the random walk started at i. x_0 is defined as the probability distribution with 1.0 at position i 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 at timestep t. PPV can also be defined as

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

Recently there has been interesting theoretical work for using random walk based approaches for computing good quality local graph partitions (cluster) near a given seed node. The quantity used for clustering is $v_i(j)/d(j)$, where $d(j)$ is the weighted degree of node j. For an undirected graph this measure is symmetric, i.e. we have $v_i(j)/d(j) = v_j(i)/d(i)$ (Appendix), and also 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 this quantity as well for our experiments and analysis. We would also like to mention that the authors in [1] used a lazy random walk. Although our random walks are not lazy, it can be shown using derivations from [1] that the personalized pagerank arising from an *un-lazy* random walk with restart probability α is equivalent to that with a lazy random walk with restart probability $\alpha/(2-\alpha)$.

Conductance of a partition (cluster) is used as a measure of its quality. 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 [15], conductance is defined as:

$$\Phi_V(A) = \frac{E(A, V-A)}{min(\mu(A), \mu(V-A))} \tag{2}$$

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 [15]. In [1] the authors improved upon the former algorithm by computing local cuts from personalized pagerank vectors from the predefined seed nodes. A random walk started inside a low conductance cluster will mostly stay inside the cluster. Hence partitions based on PPV values will automatically translate to lower number of page-faults, while simulating a short random walk.

2.2 Algorithm RWDISK-simple

We will begin will a simpler algorithm, called *RWDISK-simple* to compute PPV values from a set of anchor nodes. 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 $p = P(X_t = \texttt{destnode} | X_{t-1} = \texttt{sourcenode})$. 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

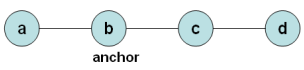


Figure 1: Line Graph

`value` equals $P(X_{t-1} = \texttt{source} | X_0 = \texttt{anchor})$. *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 $P(X_t = \texttt{source} | X_0 = \texttt{anchor})$. *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 $\texttt{value} = \sum_{n=1}^{t} \alpha(1-\alpha)^{n-1} x_{n-1}(j)$. *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 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).

2.3 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.

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` $\times$ `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 $\texttt{maxiter} \times A \times d/\epsilon_{t-1}$. Since we increase ϵ_t every iteration, $\epsilon_t \geq \epsilon, \forall t$. Here are the file-sizes obtained from RWDISK in a nutshell:

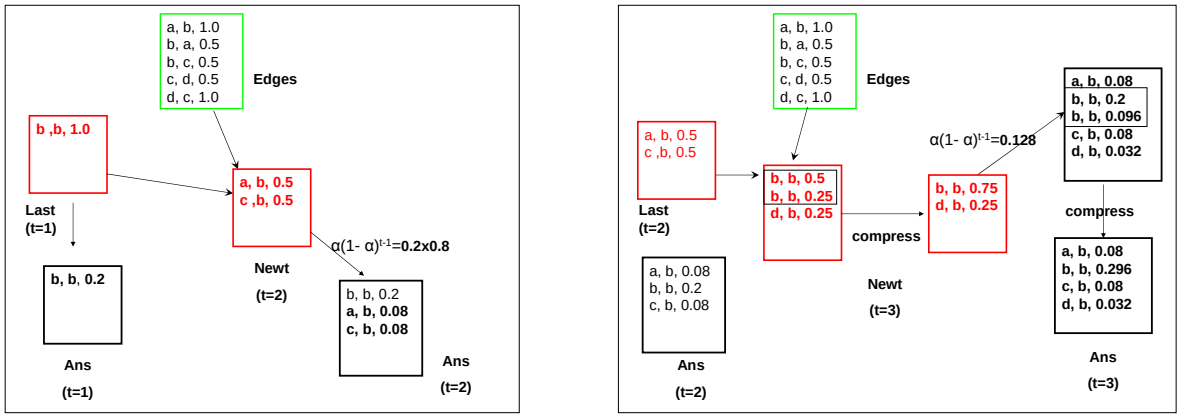
Algorithm	# anchors	*Last* Size	*Newt* Size	*Ans* Size
No-pruning	$\frac{N}{n_a}$	$O(\frac{N^2}{n_a})$	$O(\frac{N^2}{n_a})$	$O(\frac{N^2}{n_a})$
Pruning	$\frac{N}{n_a}$	$O(\frac{N}{\epsilon \times n_a})$	$O(\frac{E}{\epsilon \times n_a})$	$O(\frac{\texttt{maxiter} \times E}{\epsilon \times n_a})$

2.4 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



(A) (B)

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.

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.
- $i = 1$.
- Create Newt files from a pass over the *Last* and *Edges* files. *Newt*← **CreateNewtFile(*Last*,*Edges*)**.
- If the *Newt* file is empty or $i >$ `maxiter`, return *Ans* after sorting and compressing it.
- Update the *Ans* file. *Ans*← **Update(*Ans*,*Newt*,α,$i+1$)**.
- 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.

CreateNewtFile(*Last*,*Edges*)

- Read triplets {*src*,*anchor*,*val*} from *Last*with common prefix "src"
- Read triplets {*src*,*dst*,*prob*} from *Edges*with common prefix "src"
- Append to a temporary buffer *Tmp* the triplets {*dst*,*anchor*,*prob* × *val*}
- If number of lines in *Tmp* $\geq M$ sort and compress *Tmp*, and merge with already sorted file *Newt*

Table 1: Pseudocode for RWDISK

$\bar{\epsilon}_t$ 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 2.1. *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 $0 \leq \bar{\epsilon}_t(i) \leq xI(x \leq \epsilon_t) \leq \epsilon_t$, 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 2.2. *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 2.3. *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 2.3 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 $\frac{d(i)}{\delta}\frac{\epsilon}{1-\sqrt{1-\alpha}}$. 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 $\frac{\epsilon}{\delta(1-\sqrt{1-\alpha})}$.

In fact this result turns out to be very similar to that

of [1], 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 2.4. *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 2.3 and 2.4 to obtain the final approximation error guarantee.

Lemma 2.5. *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.

3. CLUSTERING

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-anchor,value`}, where `src` is a node which is not an orphan. `closest-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. EFFECT OF HIGH DEGREE NODES ON PERSONALIZED PAGERANK

In spite of rounding one problem with our power-iteration like algorithm 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. Although a high degree node might have a high personalized pagerank, the probability passed on to the out-neighbors of it can become very small. As a result in the rounding step most of these nodes might get pruned. Independent of that, in this section we will theoretically analyze the effect of high degree nodes on personalized pagerank in *undirected* graphs.

The main intuition behind this analysis is that a very high degree node passes on a small fraction of its value to the outneighbors, which might not be significant enough for spending 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. Also the error incurred in our preferred degree-normalized 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 will analyze the effect of making a high degree node 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. $PPV(\boldsymbol{r}, j)$ denotes the personalized pagerank value at node j w.r.t start distribution $\boldsymbol{r}$. We will denote $\boldsymbol{r_i}$ as the indicator vector for i. For $\boldsymbol{r} = \boldsymbol{r_i}$ we have personalized pagerank from node i. For PPV values from node i to j we will interchangeably use $PPV(\boldsymbol{r_i}, j)$ and $PPV(i, j)$. Let $\widehat{PPV}$ be the personalized pagerank w.r.t. start distribution r on the changed transition matrix.

Theorem 4.1. *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 4.1 we will prove that if degree of s is much higher that than of i, then the error will be small.

Lemma 4.2. *The error introduced at node $i \neq s$ by converting node s into a sink can be upper bounded by $\frac{(1-\alpha)d_i}{\alpha d_s}$.*

Hence if d_s is much larger than d_i then this error is small. Also, since we are using $PPV(i, j)/d_j$ for clustering, the error becomes $\frac{1-\alpha}{\alpha d_s}$. 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 4.3. *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 $\frac{d(i)}{\min_{s_j \in S} d(s_k)}$.*

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.

5. DETERMINISTIC ALGORITHMS WITH CLUSTERS

We can simulate random walks from a node in our clustered representation. 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 top ranked nodes in PPV w.r.t a node. 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 [8] and further developed for computing sparse personalized pagerank vectors by [12]. For a given node i, the personalized pagerank 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

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

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 $PPV(k,i) = \sum_{m\in\delta(S)} Pr(X_m|X_{\delta(S)})PPV(m,i)$, 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 probability 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:

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

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

Since S is small in size, the power method suffices for computing these bounds, one could also use rounding methods introduced by [12]. 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.

Once the 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 4 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 size of a high degree node, most nodes are far away from it leading to an uninteresting set of nearest neighbors anyways.

6. RELATION TO PREVIOUS WORK

Recently there has been interesting theoretical work for using random walk based approaches for computing good quality local graph partitions (cluster) near a given seed node. The main intuition is that if a random walk should mostly stay inside a cluster with low conductance (equation 2). The formal algorithm to compute a low conductance local partition near a pre-selected seed node was given in [15]. 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.

In [1] the authors improved upon the above result by computing local cuts from personalized pagerank vectors from the predefined seed nodes. Similar to [3], the authors obtain sparse representations of personalized pagerank vectors by adaptively picking the node with a significant amount of probability and propagating it to its neighbors. For a disk-resident graph it becomes harder to *adaptively* spread the most significant probability mass. Our technique is similar in the sense that we spread the probabilities above a threshold in batch fashion. In fact our error bounds 2.3 turns

out to be very similar to that of [1]. The authors in [12] presented a rounding and sketching algorithm for simultaneously computing and storing sparse personalized pagerank vectors from all nodes in a graph. The goal is to compute PPV values from any query node using these stored vectors. This is a promising future avenue for RWDISK, but in its current form would require processing of PPV vectors from all nodes, not just the anchors.

Clustering external memory graphs for faster query processing was presented in [6]. This was optimized for a particular use case, whereas partitions resulting from RWDISK are designed for general random walk based computations. In [13] a novel algorithm is proposed to compute a single random walk of length l using $O(\sqrt{l})$ passes. The main idea is to construct a random walk by merging together shorter random walks. While this uses only sequential scans, it incurs a big overhead for processing arbitrary queries on the fly. Although the authors present an external memory scheme, it is most efficient when the pagerank vectors are stored in memory, which we are trying to avoid.

7. RESULTS

We present our results on three real-world graphs: a connected subgraph of the Citeseer co-authorship network, the entire DBLP corpus, and Live Journal. We used an undirected graph representation, although RWDISK can be used for directed graphs as well. 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 [1]. Our results are shown in three steps: first we compare the usefulness of the clusters obtained from RWDISK and the baseline algorithm METIS. Next we present the number of pagefaults for computing nearest neighbors in an actual link prediction task on the clustered graphs. In the end we show the effect of clever deterministic algorithms for nearest-neighbor computation on reducing the total number of page-faults by cleverly bringing in the right clusters.

7.1 RWDISK and METIS

We used METIS as a baseline algorithm, which is a state of the art graph partitioning algorithm and is *not an external memory* 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. Also, 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 4 confirms that we did achieve a huge computational gain without sacrificing the quality of approximation.

Measure of cluster quality. Conductance (2) is a popular measure of cluster quality in undirected graphs. 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. However note that our concern is not only a small conductance, but also the fact that cluster sizes are not too big, because then we will not be able to store it on one disk-page. In our graph representation roughly 300 edges can be fit into a disk-page. Hence every time we see a cluster of size $300 \times k$ we will count k pagefaults to load the cluster from disk. In order to take this into account, we define a different cluster quality measure. Note that conductance c roughly measures the average number of times a random walk can escape outside a cluster. Lets say the *mean* cluster size is m. We will define our measure as $c \times \lceil m/300 \rceil$. This roughly captures the expected number of pagefaults everytime a random walk leaves a cluster. How do we compute mean size? If I randomly pick a node what will be the expected size of the cluster it belongs to? The number of directed edges of a cluster gives us the number of lines in the cluster-file. We compute the weighted mean of edges, weighed by the number of nodes in a cluster. We look at the *conductance*$\times$*meansize* values, and plot the histogram for the number of nodes in a cluster with that property in figure 3.

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 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. The buffersize was fixed at 100. 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

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

Table 2: # nodes,directed edges in graphs

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

Table 3: For each dataset, the minimum degree of a sink node, and the total number of sink nodes, time for RWDISK.

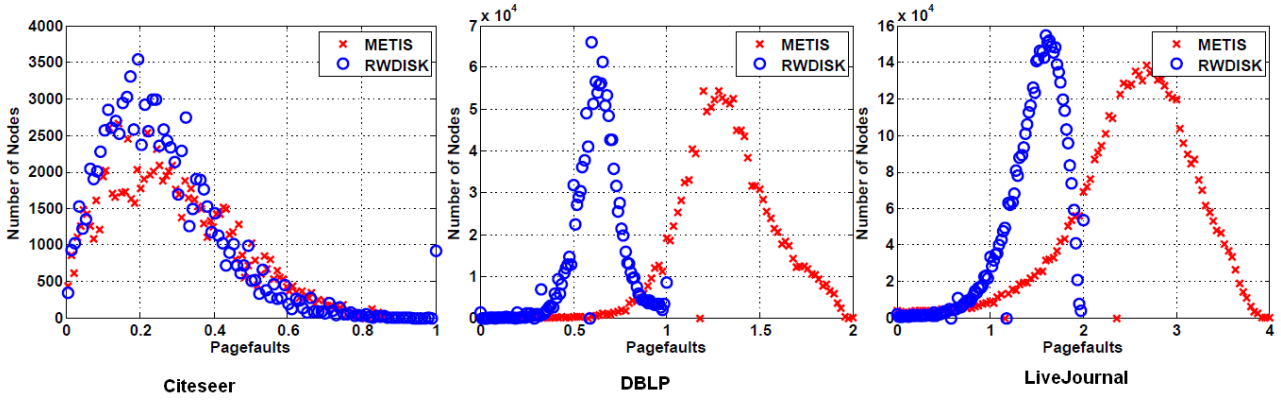


Figure 3: The histograms showing the expected number of pagefaults if a random walk stepped outside a cluster for a randomly picked node. Left to right the panels are for Citeseer, DBLP and LiveJournal respectively.

seconds is the total time taken for all 50 simulations averaged over the 100 random nodes. These numbers exactly match our expectation. We see that on Citeseer METIS and

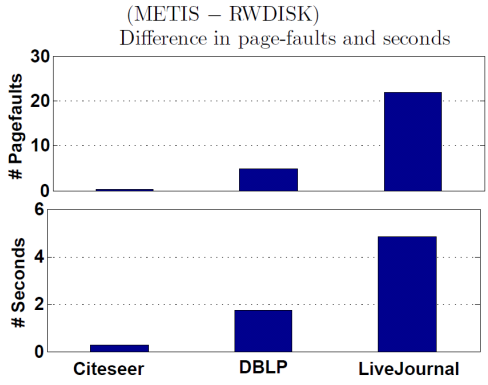


Figure 4: #Page-faults(METIS)-#Page-faults(RWDISK) per 20 step random walk in upper panel. Bottom Panel contains total time for simulating 50 such random walks. Both are averaged over 100 randomly picked source nodes.

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

7.2 Link Prediction

Converting high degree nodes into sinks can speed up the RWDISK algorithm significantly. For example, on the LiveJournal graph the RWDISK became 3.5 times as much faster by turning nodes with degree above a hundred (instead of thousand) into sinks. In this section we will show the effect of this process on link prediction. Intuitively by making a node into sink, we are saving computation without hurting the approximation quality too much. Surprisingly though, in our experiments we saw that link-prediction scores went up after converting high degree nodes into sinks. 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 help-out neighbors in top 10 ranked nodes in degree-normalized personalized pagerank from i. Only nodes below degree 100 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. We used a size 100 buffer and the *least recently used* replacement scheme. Each time a random walk moved to a cluster not already present in the buffer, we incremented the number of pagefaults by $k = \lceil \frac{e}{300} \rceil$, where e is total number of directed edges (number of lines in the file), since an ideal page-sized file should contain about 300 lines in our representation. We would also like to point out that since this is not AUC score, a random prediction does much worse that 0.5 in these tasks. Here is the reason behind 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

above 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. But it is hard to believe that high degree nodes cannot be useful. In DBLP high degree nodes without exception end up being words which can confuse random walks. The LiveJournal graph has a small number of closed triangles [10] which means there are nodes which have connections with many other nodes, which might not know each other. 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 by a very slight amount once we introduce sink nodes.

7.3 Deterministic Algorithm

We present the mean and median number of pagefaults incurred by the deterministic algorithm in section 5. We executed the algorithm for computing top 10 neighbors for 500 randomly picked nodes. We used approximation slack of 0.005 for ranking. For Citeseer we computed the nearest

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

neighbors in 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 notable 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 this happens for high degree nodes, from which all other nodes are more or less farther away. Also for these nodes, the clusters have a lot of boundary nodes and hence the bounds are hard to tighten. These results show the superiority of the deterministic algorithm over random simulations in terms of number of page-faults.

8. CONCLUSION

This paper has introduced an algorithm that uses only sequential disk access for organizing a large graph on disk. The organizing principle is to partition the graph into clusters. Each cluster is associated with an anchor node. The nodes in the cluster of anchor node a are those which are "closer" to a than any other anchor. Computing these clusters requires no main memory representation of the set of edges or set of nodes in the full graph, but instead uses various merge and sort operations. The resulting clusters allow operations which need access to local elements in the graph, such as (random walk simulation) to happen with relatively few disk seeks. We have shown, via experiments with a simple ASCII-file-based implementation, that this algorithm is practical, and gives better clusters (in terms of conductance and wall-clock-time for random-walk-based queries) than METIS, the popular in-memory graph partitioning algorithm. Finally we have proved that the approximation of rounding small probabilities to zero leads to a bounded error, and that converting high degree nodes into sinks can lead to drastic improvement in performance without compromising accuracy. 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.

9. APPENDIX

Proof of symmetry of degree normalized personalized pagerank in undirected graphs. This follows directly from the reversibility of random walks.

$$v_i(j) = \alpha \sum_{t=0}^{\infty}(1-\alpha)^t P^t(i,j) = d(j)/d(i)\alpha \sum_{t=0}^{\infty}(1-\alpha)^t P^t(j,i)$$
$$\implies v_i(j)/d(j) = v_j(i)/d(i) \tag{3}$$

Proof of Lemma 2.1. 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 2.2. 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 2.3. By plugging in the result from lemma 2.2 into equation (1) the total error incurred in the PPV value is

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

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.

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

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 2.4. 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 2.5. Let $\hat{v}$ be rounded PPV values when `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 2.4 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 2.3 gives the desired result.

Proof of theorem 4.1. Personalized pagerank of a start distribution $\boldsymbol{r}$ can be written as

$$\begin{aligned} &PPV(\boldsymbol{r}) = \alpha(I - (1-\alpha)P^T)^{-1}\boldsymbol{r} = \alpha\boldsymbol{r} + \alpha\textstyle\sum_{t=1}^{\infty}(1-\alpha)^t(P^T)^t\boldsymbol{r} \\ &= \alpha\boldsymbol{r} + (1-\alpha)PPV(P^T\boldsymbol{r}) \end{aligned}$$

By turning node s into a sink, we are *only* changing the s^{th} row of P. We denote by $\boldsymbol{r_s}$ the indicator vector for node s. Essentially we are subtracting the entire row $P(s,:) = P^T\boldsymbol{r_s}$ and adding back $\boldsymbol{r_s}$. This is equivalent to subtracting the matrix $\boldsymbol{v}\boldsymbol{u}^T$ from P, where $\boldsymbol{v}$ is $\boldsymbol{r_s}$ and $\boldsymbol{u}$ is defined as $P^T\boldsymbol{r_s} - \boldsymbol{r_s}$.

$$PPV(\boldsymbol{r}) = \alpha(I - (1-\alpha)P^T + (1-\alpha)\boldsymbol{u}\boldsymbol{v}^T)^{-1}\boldsymbol{r}$$

We will use the Sherman-Morrison formula [14] to compute the difference between $PPV(\boldsymbol{r})$ and $\widehat{PPV}(\boldsymbol{r})$. We will use M to denote $I-(1-\alpha)P^T$. Hence $PPV(\boldsymbol{r}) = \alpha M^{-1}\boldsymbol{r}$. A straightforward application of the Sherman Morrison lemma gives

$$\begin{aligned} \widehat{PPV}(\boldsymbol{r}) &= \alpha[M + (1-\alpha)\boldsymbol{u}\boldsymbol{v}^T]^{-1}\boldsymbol{r} \\ &= \alpha[M^{-1} - (1-\alpha)\frac{M^{-1}\boldsymbol{u}\boldsymbol{v}^T M^{-1}}{1 + (1-\alpha)\boldsymbol{v}^T M^{-1}\boldsymbol{u}}]\boldsymbol{r} \\ &= PPV(\boldsymbol{r}) - \alpha(1-\alpha)\frac{M^{-1}\boldsymbol{u}\boldsymbol{v}^T M^{-1}}{1 + (1-\alpha)\boldsymbol{v}^T M^{-1}\boldsymbol{u}}\boldsymbol{r} \end{aligned}$$

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 $PPV(\boldsymbol{r}, s)$ So the numerator of the difference becomes

$$(1-\alpha)/\alpha[PPV(P^T\boldsymbol{r_s}) - PPV(r_s)]PPV(\boldsymbol{r}, s)$$

From the definition of PPV in equation (4) we can further simplify the numerator to be $[PPV(\boldsymbol{r_s}) - \boldsymbol{r_s}]PPV(\boldsymbol{r}, s)$. Now for the denominator we have

$$\begin{aligned} &1 + (1-\alpha)\boldsymbol{v}^T M^{-1}\boldsymbol{u} \\ &= 1 + \tfrac{1-\alpha}{\alpha}\boldsymbol{r_s}^T(PPV(P^T\boldsymbol{r_s}) - PPV(r_s)) \\ &= 1 + \boldsymbol{r_s}^T[PPV(\boldsymbol{r_s}) - \boldsymbol{r_s}] = PPV(\boldsymbol{r_s}, s) \end{aligned}$$

Combining the numerator and the denominator we get

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

This leads to the element-wise error bound in theorem 4.1.

Proof of lemma 4.3. 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 $\{s_1, ..., s_j\}$. 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 $G \setminus S[j] = (G \setminus S[j+1]) \setminus S[j+1]$. Which leads to:

$$\begin{aligned} &PPV^{G\setminus S[k-1]}(\boldsymbol{r}, i) - PPV^{G\setminus S[k]}(\boldsymbol{r}, i) \\ &\leq \tfrac{PPV^{G\setminus S[k-1]}(s_k, i)PPV^{G\setminus S[k-1]}(r, s_k)}{\alpha} \end{aligned}$$

Now by using a telescoping sum we have

$$\begin{aligned} &PPV^{G}(\boldsymbol{r}, i) - PPV^{G\setminus S[k]}(\boldsymbol{r}, i) \\ &\leq \textstyle\sum_{j=1}^{k} \tfrac{PPV^{G\setminus S[j-1]}(s_j, i)PPV^{G\setminus S[j-1]}(r, s_j)}{\alpha} \end{aligned}$$

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 $PPV^{G\setminus S[k-1]}(r, s_k)$ by $PPV^G(r, s_k)$. This gives us the following sum, which can be simplified using (3) as,

$$\begin{aligned} &\sum_{j=1}^{k} PPV(s_j, i)\frac{PPV(\boldsymbol{r}, s_j)}{\alpha} = \sum_{j=1}^{k} \frac{d(i)PPV(i, s_j)}{d(s_j)}\frac{PPV(\boldsymbol{r}, s_j)}{\alpha} \\ &\leq \frac{1}{\alpha}\frac{d(i)}{\min_{s_j \in S}} \sum_{j=1}^{k} PPV(i, s_j)PPV(\boldsymbol{r}, s_j) \leq \frac{1}{\alpha}\frac{d(i)}{\min_{s_j \in S}} \end{aligned}$$

The last step follows from the fact that $PPV(r, s_j) \leq 1$ and $PPV(i, s_j)$, summed over j has to be smaller than one. This leads to the final result in lemma 4.3.

10. REFERENCES

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

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

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

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. WWW*, 1998.

[5] S. Chakrabarti, J. Mirchandani, and A. Nandi. Spin: searching personal information networks. In *SIGIR '05*, 2005.

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

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

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

[9] G. Jeh and J. Widom. Simrank: A measure if structural-context similarity. In *ACM SIGKDD*, 2002.

[10] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. 2008.

[11] P. Sarkar and A. Moore. A tractable approach to finding closest truncated-commute-time neighbors in large graphs. In *Proc. UAI*, 2007.

[12] T. Sarlós, A. A. Benczúr, K. Csalogány, D. Fogaras, and B. Rácz. To randomize or not to randomize: space optimal summaries for hyperlink analysis. In *WWW '06*.

[13] A. D. Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. In *PODS*, 2008.

[14] J. Sherman and W. J. Morrison. Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix (abstract). *Annals of Mathematical Statistics*, 1949.

[15] D. Spielman and S. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the STOC'04*, 2004.