Exploiting the Block Structure of the Web for Computing PageRank

Sepandar D. Kamvar Taher H. Haveliwala Christopher D. Manning Gene H. Golub

Stanford University
{sdkamvar,taherh,manning,golub}@cs.stanford.edu

Abstract

The web link graph has a nested block structure: the vast majority of hyperlinks link pages on a host to other pages on the same host, and many of those that do not link pages within the same domain. We show how to exploit this structure to speed up the computation of PageRank by a 3-stage algorithm whereby (1) the local PageRanks of pages for each host are computed independently using the link structure of that host, (2) these local PageRanks are then weighted by the "importance" of the corresponding host, and (3) the standard PageRank algorithm is then run using as its starting vector the weighted concatenation of the local PageRanks. Empirically, this algorithm speeds up the computation of PageRank by a factor of 2 in realistic scenarios. Further, we develop a variant of this algorithm that efficiently computes many different "personalized" PageRanks, and a variant that efficiently recomputes PageRank after node updates.

1 Introduction

The rapidly growing web graph contains several billion nodes, making graph-based computations very expensive. One of the best known web-graph computations is PageRank, an algorithm for determining the "importance" of Web pages [14]. The core of the PageRank algorithm involves repeatedly iterating over the web graph structure until a stable assignment of page-importance estimates is obtained. As this computation can take several days on web graphs of several billion nodes, the development of techniques for reducing the computational costs of the algorithm becomes necessary for two reasons. Firstly, speeding up this computation is part of the general efficiencies needed for improving the freshness of a web index. Secondly, recent approaches to personalized and topic-sensitive PageRank schemes [8, 10, 16] require computing *many* PageRank vectors, intensifying the need for faster methods for computing PageRank.

Previous approaches to accelerating the PageRank computation exploit general sparse graph techniques. Arasu et al. [1] use values from the current iteration as they become available, rather than using only values from the previous iteration. They also suggest that exploiting the "bow-tie" structure of the web [3] would be useful in computing PageRank. However, the method they suggest is not practical, as ordering the web matrix according to this structure requires depth-first search, which is prohibitively costly on the web. More recently, Kamvar et al. [11] suggest using successive intermediate iterates to extrapolate successively better estimates of the true PageRank values. However, the speedups from the latter work are modest for the parameter settings typically used for PageRank.[1]

This paper proposes exploiting the detailed typology of the web graph. Analysis of the graph structure of the web has concentrated on determining various properties of the graph, such as degree distributions and connectivity statistics, and on modeling the creation of the web graph [12, 2]. However, this research has not directly addressed how this inherent structure can be exploited effectively to speed up link analysis. Raghavan and Garcia-Molina [15] have exploited the hostname (or more generally, url)-induced structure of the web to represent the web graph efficiently. In this paper, we directly exploit this kind of structure to achieve large speedups compared with previous algorithms for computing PageRank by

- Substantially improving locality of reference, thereby reducing disk i/o costs and memory access costs,
- Reducing the computational complexity (i.e., number of FLOPS).
- Allowing for highly parallelizable computations requiring little communication overhead,
- Allowing reuse of previous computations when updating PageRank and when computing multiple "personalized" PageRank vectors.

[1] In particular, the speedup is modest for the typically cited damping factor of $c = 0.85$ (which gives a fast-mixing graph).

Term	Example: cs.stanford.edu/research/
top level domain	edu
domain	stanford.edu
hostname	cs
host	cs.stanford.edu
path	/research/

Table 1: Example illustrating our terminology using the sample url http://cs.stanford.edu/research/.

		Domain		Host	
Full	*Intra*	953M links	83.9%	899M links	79.1%
	Inter	183M links	16.1%	237M links	20.9%
DNR	*Intra*	578M links	95.2%	568M links	93.6%
	Inter	29M links	4.8%	39M links	6.4%

Table 2: Hyperlink statistics on LARGEWEB for the full graph (*Full*: 291M nodes, 1.137B links) and for the graph with dangling nodes removed (*DNR*: 64.7M nodes, 607M links).

2 Experimental Setup

In the following sections, we investigate the structure of the web, introducing an algorithm for computing PageRank, and discuss the speedup this algorithm achieves on realistic datasets. Our experimental setup is as follows.

We used two datasets of different sizes for our experiments. The STANFORD/BERKELEY link graph was generated from a crawl of the stanford.edu and berkeley.edu domains created in December 2002 by the Stanford WebBase project. This link graph (after dangling node removal, discussed below) contains roughly 683,500 nodes, with 7.6 million links, and requires 25MB of storage. We used STANFORD/BERKELEY while developing the algorithms, to get a sense for their performance. For real-world performance measurements, we use the LARGEWEB link graph, generated from a crawl of the Web that had been created by the Stanford WebBase project in January 2001 [9]. The full version of this graph, termed FULL-LARGEWEB, contains roughly 290M nodes, just over a billion edges, and requires 6GB of storage. Many of these nodes are dangling nodes (pages with no outlinks), either because the pages genuinely have no outlinks, or because they are pages that have been discovered but not crawled. In computing PageRank, these nodes are excluded from the web graph until the final few iterations, so we also consider the version of LARGEWEB with dangling nodes removed, termed DNR-LARGEWEB, which contains roughly 70M nodes, with over 600M edges, and requires 3.6GB of storage. The link graphs are stored using an adjacency list representation, with pages represented as 4-byte integer identifiers. On an AMD Athlon 1533MHz machine with a 6-way RAID-5 disk volume and 3.5GB of main memory, each iteration of PageRank on the 70M page DNR-LARGEWEB dataset takes roughly 7 minutes. Given that computing PageRank generally requires up to 100 iterations, the need for fast computational methods for larger graphs with billions of nodes is clear.

Our criterion for determining the convergence of the algorithms that follow uses the L_1 norm of the residual vector; i.e., $||A\vec{x}^{(k)} - \vec{x}^{(k)}||_1$. We refer the reader to [11] for a discussion of why the L_1 residual is an appropriate measure for measuring convergence.

3 Block Structure of the Web

The key terminology we use in the remaining discussion is given in Table 1.

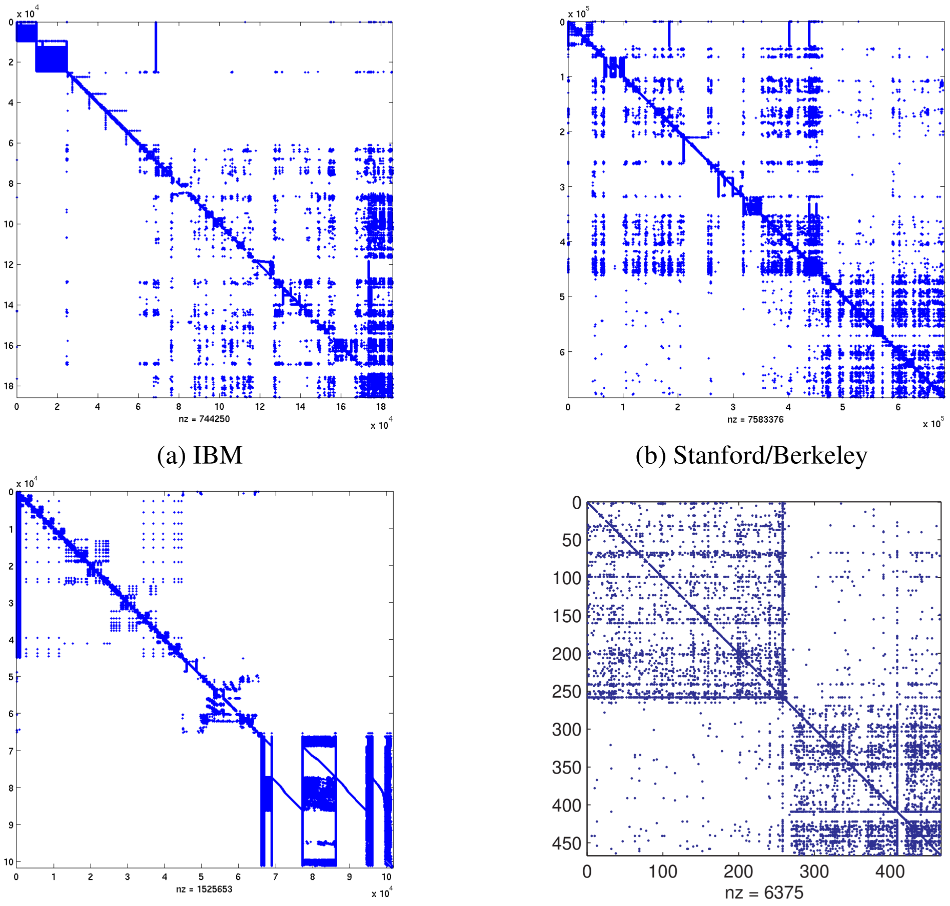
To investigate the structure of the web, we run the following simple experiment. We take all the hyperlinks in FULL-LARGEWEB, and count how many of these links are "intra-host" links (links from a page to another page in the same host) and how many are "inter-host" links (links from a page to a page in a different host). Table 2 shows that 79.1% of the links in this dataset are intra-host links, and 20.9% are inter-host links. These intra-host connectivity statistics are not far from the earlier results of Bharat et al. [2]. We also investigate the number of links that are intra-domain links, and the number of links that are inter-domain links. Notice in Table 2 that an even larger majority of links are intra-domain links (83.9%).

These results make sense intuitively. Take as an example the domain cs.stanford.edu. Most of the links in cs.stanford.edu are links around the cs.stanford.edu site (such as cs.stanford.edu/admissions, or cs.stanford.edu /research). Furthermore, almost all non-navigational links are links to other Stanford hosts, such as www.stanford.edu, library.stanford.edu, or www-cs-students.stanford.edu.

One might expect that there exists this structure even in lower levels of the web hierarchy. For example, one might expect that pages under cs.stanford.edu/admissions/ are highly interconnected, and very loosely connected with pages in other sublevels, leading to a nested block structure. This type of nested block structure can be naturally exposed by sorting the link graph to construct a link matrix in the following way. We sort urls lexicographically, except that as the sort key, we reverse the components of the domain. For instance, the sort key for the url www.stanford.edu/home/students/ would be edu.stanford.www/home/students. The urls are then assigned sequential identifiers when constructing the link matrix. A link matrix contains as its (i, j)th entry a 1 if there is a link from i to j, and 0 otherwise. This has the desired property that urls are grouped in turn by top level domain, domain, hostname, and finally path. The subgraph for pages in stanford.edu appear as a sub-block of the full link matrix. In turn, the subgraph for pages in www-db.stanford.edu appear as a nested sub-block.

We can then gain insight into the structure of the web by using dotplots to visualize the link matrix. In a dotplot, if there exists a link from page i to page j then point (i, j) is colored; otherwise, point (i, j) is white. Since our full datasets are too large to see individual pixels, we show several slices of the web in Figure 1. Notice three things:

1. There is a definite block structure to the web.
2. The individual blocks are much smaller than entire web.
3. There are clear nested blocks corresponding to domains,



(a) IBM

(b) Stanford/Berkeley

(c) Stanford-50

(d) Stanford/Berkeley Host Graph

Figure 1: A view of 4 different slices of the web: (a) the IBM domain, (b) all of the hosts in the Stanford and Berkeley domains, (c) the first 50 Stanford domains, alphabetically, and (d) the host-graph of the Stanford and Berkeley domains.

hosts, and subdirectories within the path.

Figure 1(a) shows the dotplot for the ibm.com domain. Notice that there are clear blocks, which correspond to different hosts within ibm.com; for example, the upper left block corresponds to the almaden.ibm.com hosts (the hosts for IBM's Almaden Research Center). Notice that the pages at the very end of the plot (pages $i \geq 18544$) are heavily inlinked (the vertical line at the lower right hand corner of the plot. These are the pages within the www.ibm.com host, and it is expected that they be heavily inlinked. Also notice that there are 4 patterns that look like the upside-down letter "L". These are sites that have a shallow hierarchy; the root node links to most pages in the host (horizontal line), and is linked to by most pages in the host (vertical line), but there is not much interlinkage within the site itself (empty block). Finally, notice that the area around the diagonal is very dense; this corresponds to strong intrablock linkage, especially in the smaller blocks.

Figure 1(b) shows the dotplot for STANFORD/BERKELEY. Notice that this also has a strong block structure and a dense diagonal. Furthermore, this plot makes clear the nested block structure of the web. The superblock on the upper left hand side is the stanford.edu domain, and the superblock on the lower right hand side is the berkeley.edu domain.

Figure 1(c) shows a closeup of the first 50 hosts alphabetically within the stanford.edu domain. The majority of this dotplot is composed of 3 hosts that are large: acomp.stanford.edu, the academic computing site at Stanford, in the upper left hand corner; cmgm.stanford.edu, an online bioinformatics resource, in the middle, and daily.stanford.edu, the web site for the *Stanford Daily* (Stanford's student newspaper) in the lower right hand corner. There are many interesting structural motifs in this plot. First there is a long vertical line in the upper left-hand corner. This feature corresponds to the web site http://acomp.stanford.edu; most pages in the acomp.stanford.edu host point to this root node. Also, there is a clear nested block structure within acomp.stanford.edu on the level of different directories in the url hierarchy.

In the *Stanford Daily* site, we see diagonal lines, long vertical blocks, a main center block, and short thin blocks. The first several web pages in daily.stanford.edu represent the front page of the paper for the past several days. Each front page links to the front page of the day before, and therefore there is a small diagonal line in the upper left hand corner of the *Stanford Daily* block. The diagonals are due to the url naming convention of the *Stanford Daily* which causes the lexicographic ordering of urls to induce a chronological ordering of the articles. The front pages link to articles, which are the middle pages of the block. Therefore, we see a horizontal strip in the top middle. These articles also link back to the front pages, and so we see a vertical strip on the middle left hand side. The articles link to each other, since each article links to related articles and articles by the same author. This accounts for the square block in the center. The long vertical strips represent pages that are on the standard menu on each page of the site (some pages on this menu are the "subscriptions" page, the "write a letter to the editor" page, and the "advertising" page). Finally, the diagonal lines that surround the middle block are pages such as "e-mail this article to a friend" or "comment on this article", that are linked to only one article each.

Figure 1(d) shows the host graph for the stanford.edu and berkeley.edu domains, in which each host is treated as a single node, and an edge is placed between host i and host j if there is a link between any page in host i and host j. Again, we see strong block structure on the domain level, and the dense diagonal shows strong block structure on the host level as well. The vertical and horizontal lines near the bottom right hand edge of both the Stanford and Berkeley domains represent the www.stanford.edu and www.berkeley.edu hosts, showing that these hosts are, as expected, strongly linked to most other hosts within their own domain.

3.1 Block Sizes

We investigate here the sizes of the hosts in the web. Figure 2(a) shows the distribution over number of (crawled) pages of the hosts in LARGEWEB. Notice that the majority of sites are small, on the order of 10^0 pages. Figure 2(b) shows the sizes of the host blocks in the web when dangling nodes are removed. When dangling nodes are removed, the blocks become smaller, and the distribution is still skewed towards small blocks. The largest block had 6,000 pages. In future sections we see how to exploit the small sizes of the blocks, relative to the dataset as a whole, to speedup up link analysis.

3.2 The GeoCities Effect

While one would expect that most domains have high intracluster link density, as in cs.stanford.edu, there are some domains that one would expect to have low intracluster link density, for example pages.yahoo.com (formerly www.geocities.com). The web site http://pages.yahoo.com is the root page for Yahoo! GeoCities, a free web hosting service. There is no reason to think that people who have web sites on GeoCities would prefer to link to one another rather than to sites not in GeoCities.[2] Figure 3 shows a histogram of the intradomain densities of the web. In Figure 3(a) there is a spike near 0% intrahost linkage, showing that many hosts are not very interconnected. However, when we remove the hosts that have only 1 page (Figure 3(b)), this spike is substantially dampened, and when we exclude hosts with fewer than 5 pages, the spike is eliminated. This shows that the hosts in LARGEWEB that are not highly intraconnected are very small in size. When the very small hosts are removed, the great majority of hosts have high intra-host densities, and very few hosts suffer from the GeoCities effect.

[2] There may of course be deeper structure found in the path component, although we currently do not directly exploit such structure.

4 BlockRank Algorithm

We now present the BlockRank algorithm that exploits the empirical findings of the previous section to speed up the computation of PageRank. This work is motivated by and builds on aggregation/disaggregation techniques [5, 17] and domain decomposition techniques [6] in numerical linear algebra. Steps 2 and 3 of the BlockRank algorithm are similar to the Rayleigh-Ritz refinement technique [13]. We begin with a review of PageRank in Section 4.1.

4.1 Preliminaries

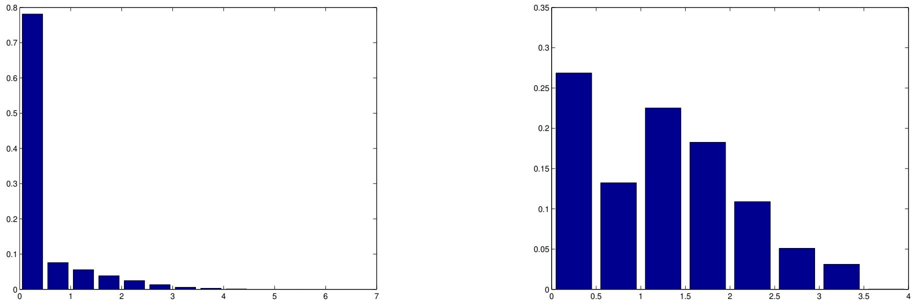
In this section we summarize the definition of PageRank [14] and review some of the mathematical tools we will use in analyzing and improving the standard iterative algorithm for computing PageRank.

Underlying the definition of PageRank is the following basic assumption. A link from a page $u \in Web$ to a page $v \in Web$ can be viewed as evidence that v is an "important" page. In particular, the amount of importance conferred on v by u is proportional to the importance of u and inversely proportional to the number of pages u points to. Since the importance of u is itself not known, determining the importance for every page $i \in Web$ requires an iterative fixed-point computation.

To allow for a more rigorous analysis of the necessary computation, we next describe an equivalent formulation in terms of a random walk on the directed Web graph G. Let $u \rightarrow v$ denote the existence of an edge from u to v in G. Let $\deg(u)$ be the outdegree of page u in G. Consider a random surfer visiting page u at time k. In the next time step, the surfer chooses a node v_i from among u's outneighbors $\{v|u \rightarrow v\}$ uniformly at random. In other words, at time $k+1$, the surfer lands at node $v_i \in \{v|u \rightarrow v\}$ with probability $1/\deg(u)$.

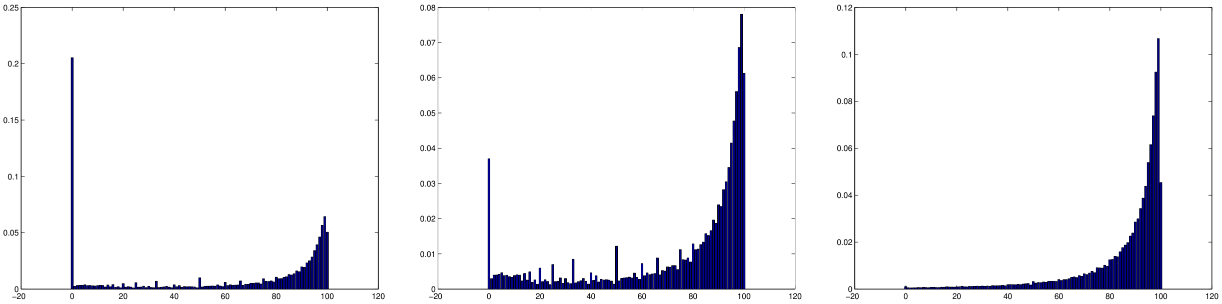
The PageRank of a page i is defined as the probability that at some particular time step $k > K$, the surfer is at page i. For sufficiently large K, and with minor modifications to the random walk, this probability is unique, illustrated as follows. Consider the Markov chain induced by the random walk on G, where the states are given by the nodes in G, and the stochastic transition matrix describing the transition from i to j is given by P with $P_{ij} = 1/\deg(i)$.

For P to be a valid transition probability matrix, every node must have at least 1 outgoing transition; e.g., P should have no rows consisting of all zeros. This property holds



(a) FULL-LARGEWEB (b) DNR-LARGEWEB

Figure 2: Histogram of distribution over host sizes of the web. The x-axis gives bucket sizes for the $\log_{10}$ of the size of the host-blocks, and the y-axis gives the fraction of host-blocks that are that size.



(a) All hosts (b) Hosts with 1 page excluded (c) Hosts with < 5 pages excluded

Figure 3: Distribution over interconnectivity of host blocks for the DNR-LARGEWEB data set. The x-axis of each figure shows percentile buckets for intra-host linkage density (the percent of edges originating or terminating in a given host that are intra-host links), and the y-axis shows the fraction of hosts that have that linkage density. Figure 3(a) shows the distribution of intra-host linkage density for all hosts; 3(b) shows it for all hosts that have more than 1 page; and 3(c) shows it for all hosts that have 5 or more pages.

if G does not have any pages with outdegree 0, which does not hold for the Web graph. P can be converted into a valid transition matrix by adding a complete set of outgoing transitions to pages with outdegree 0. In other words, we can define the new matrix P' where all states have at least one outgoing transition in the following way. Let n be the number of nodes (pages) in the Web graph. Let $\vec{v}$ be the n-dimensional column vector representing a uniform probability distribution over all nodes:

$$\vec{v} = [\frac{1}{n}]_{n \times 1} \tag{1}$$

Let $\vec{d}$ be the n-dimensional column vector identifying the nodes with outdegree 0:

$$d_i = \begin{cases} 1 & \text{if } \deg(i) = 0, \\ 0 & \text{otherwise} \end{cases}$$

Then we construct P' as follows:

$$D = \vec{d} \cdot \vec{v}^T$$
$$P' = P + D$$

In terms of the random walk, the effect of D is to modify the transition probabilities so that a surfer visiting a dangling page (i.e., a page with no outlinks) randomly jumps to a page in the next time step, using the distribution given by $\vec{v}$.

By the Ergodic Theorem for Markov chains [7], the Markov chain defined by P' has a unique stationary probability distribution if P' is aperiodic and irreducible; the former holds for the Markov chain induced by the Web graph. The latter holds iff G is strongly connected, which is generally *not* the case for the Web graph. In the context of computing PageRank, the standard way of ensuring this property is to add a new set of complete outgoing transitions, with small transition probabilities, to *all* nodes, creating a complete (and thus strongly connected) transition graph. In matrix notation, we construct the irreducible Markov matrix P'' as follows:

$$E = [1]_{n \times 1} \times \vec{v}^T$$
$$P'' = cP' + (1 - c)E$$

In terms of the random walk, the effect of E is as follows. At each time step, with probability $(1 - c)$, a surfer visiting any node will jump to a random Web page (rather than following an outlink). The destination of the random jump is chosen according to the probability distribution given in $\vec{v}$. Artificial jumps taken because of E are referred to as *teleportation*. In [14], the value of c was taken to be 0.85, and we use this value in all experiments in this work.

By redefining the vector $\vec{v}$ given in Equation 1 to be nonuniform, so that D and E add artificial transitions with nonuniform probabilities, the resultant PageRank vector

$\vec{y} = cP^T\vec{x};$
$w = ||\vec{x}||_1 - ||\vec{y}||_1;$
$\vec{y} = \vec{y} + w\vec{v};$

Algorithm 1: Computing $\vec{y} = A\vec{x}$

function **pageRank**$(G, \vec{x}^{(0)}, \vec{v})$ {
Construct P from G: $P_{ji} = 1/\deg(j)$;
repeat
$\quad \vec{x}^{(k+1)} = cP^T\vec{x}^{(k)};$
$\quad w = ||\vec{x}^{(k)}||_1 - ||\vec{x}^{(k+1)}||_1;$
$\quad \vec{x}^{(k+1)} = \vec{x}^{(k+1)} + w\vec{v};$
$\quad \delta = ||\vec{x}^{(k+1)} - \vec{x}^{k}||_1;$
until $\delta < \epsilon$;
return $\vec{x}^{(k+1)}$;
}

Algorithm 2: PageRank

can be biased to prefer certain kinds of pages. For this reason, we refer to $\vec{v}$ as the *personalization* vector.

For simplicity and consistency with prior work, the remainder of the discussion will be in terms of the transpose matrix, $A = (P'')^T$; i.e., the transition probability distribution for a surfer at node i is given by row i of P'', and column i of A.

Note that the edges artificially introduced by D and E never need to be explicitly materialized, so this construction has no impact on efficiency or the sparsity of the matrices used in the computations. In particular, the matrix-vector multiplication $\vec{y} = A\vec{x}$ can be implemented efficiently using Algorithm 1.

Assuming that the probability distribution over the surfer's location at time 0 is given by $\vec{x}^{(0)}$, the probability distribution for the surfer's location at time k is given by $\vec{x}^{(k)} = A^k\vec{x}^{(0)}$. The unique stationary distribution of the Markov chain is defined as $\lim_{k\to\infty} x^{(k)}$, which is equivalent to $\lim_{k\to\infty} A^k x^{(0)}$, and is independent of the initial distribution $\vec{x}^{(0)}$. This stationary distribution is simply the principal eigenvector of the matrix $A = (P'')^T$, which is the PageRank vector we would like to compute.

The standard PageRank algorithm computes the principal eigenvector by starting with the uniform distribution $\vec{x}^{(0)} = \vec{v}$ and computing successive iterates $\vec{x}^{(k+1)} = A\vec{x}^{(k)}$ until convergence (i.e., it uses the *power method*). This algorithm is summarized in Algorithm 2. While many algorithms have been developed for fast eigenvector computations, many of them are unsuitable for this problem because of the size and sparsity of the Web matrix (see [11] for discussion).

4.2 Overview of BlockRank Algorithm

The block structure of the web suggests a fast algorithm for computing PageRank, wherein a "local PageRank vector" is computed for each host, giving the relative importance of pages within a host. These local PageRank vectors can then be used to form an approximation to the global PageRank vector that is used as a starting vector for the standard PageRank computation. This is the basic idea behind the BlockRank algorithm, which we summarize here and describe in this section. The algorithm proceeds as follows:

0. Split the web into blocks by domain.
1. Compute the Local PageRanks for each block (Section 4.3).
2. Estimate the relative importance, or "BlockRank" of each block (Section 4.4).
3. Weight the Local PageRanks in each block by the BlockRank of that block, and concatenate the weighted Local PageRanks to form an approximate Global PageRank vector $\vec{z}$ (Section 4.5).
4. Use $\vec{z}$ as a starting vector for standard PageRank (Section 4.6).

We describe the steps in detail below, and we introduce some notation here. We will use lower-case indices (i.e. i, j) to represent indices of individual web sites, and upper case indices (i.e. I, J) to represent indices of blocks. We use the shorthand notation $i \in I$ to denote that page $i \in$ block I. The number of elements in block J is denoted n_J. The graph of a given block J is given by the $n_J \times n_J$ submatrix G_{JJ} of the matrix G.

4.3 Computing Local PageRanks

In this section, we describe computing a "local PageRank vector" for each block in the web. Since most blocks have very few links in and out of the block as compared to the number of links within the block, it seems plausible that the relative rankings of most of the pages within a block are determined by the inter-block links.

We define the *local PageRank vector* $\vec{l}_J$ of a block J (G_{JJ}) to be the result of the PageRank algorithm applied only on block J, as if block J represented the entire web, and as if the links to pages in other blocks did not exist.

That is:

$$\vec{l}_J = \textit{pageRank}(G_{JJ}, \vec{s}_J, \vec{v}_J)$$

where the start vector $\vec{s}_J$ is the $n_J \times 1$ uniform probability vector over pages in block J ($[\frac{1}{n_J}]_{n\times 1}$), and the personalization vector $\vec{v}_J$ is the $n_J \times 1$ vector whose elements are all zero except the element corresponding to the root node of block J, whose value is 1.

4.3.1 Local PageRank accuracies

To investigate how well these local PageRank vectors approximate the relative magnitudes of the true PageRank vectors within a given host, we run the following experiment. We compute the local PageRank vectors $\vec{l}_J$ of each host in STANFORD/BERKELEY. We also compute the global PageRank vector $\vec{x}$ for STANFORD/BERKELEY using the standard PageRank algorithm whose personalization vector $\vec{v}$ is a uniform distribution over root nodes. We then compare the local PageRank scores of the pages within a given host to the global PageRank scores of the pages in the same host.

Approximation	Error Measure	Average Value
$\vec{l}_J$	$\|\|\vec{l}_J - \vec{g}_J\|\|_1$	0.2383
	$\mathrm{KDist}(\vec{l}_J, \vec{g}_J)$	0.0571
$\vec{v}_J$	$\|\|\vec{v}_J - \vec{g}_J\|\|_1$	1.2804
	$\mathrm{KDist}(\vec{v}_J, \vec{g}_J)$	0.8369

Table 3: The "closeness" as measured by average (a) absolute error, and (b) KDist distance of the local PageRank vectors $\vec{l}_J$ and the global PageRank segments $\vec{g}_J$, compared to the closeness between uniform vectors $\vec{v}_J$ to the global PageRank segments $\vec{g}_J$ for the STANFORD/BERKELEY dataset.

Specifically, we take the elements corresponding to the pages in host J of the global PageRank vector $\vec{x}$, and form the vector $\vec{g}_J$ from these elements. We normalize $\vec{g}_J$ so that its elements sum to 1 in order to compare it to the local PageRank vector $\vec{l}_J$, which also has an L_1 norm of 1. Specifically,

$$\vec{g}_J = \vec{x}(j \in J)/||\vec{x}(j \in J)||_1$$

We call these vectors $\vec{g}_J$ normalized global PageRank segments, or simply *global PageRank segments* for short.

The results are summarized in Table 3. The absolute error $||\vec{l}_J - \vec{g}_J||_1$ is on average 0.2383 for the hosts in STANFORD/BERKELEY.

We compare the error of the local PageRank vectors $\vec{l}_j$ to the error of a uniform $\vec{v}_J = [\frac{1}{n_J}]_{n \times 1}$ vector for each host J. The error $||\vec{v}_J - \vec{g}_J||_1$ is on average 1.2804 for the hosts in STANFORD/BERKELEY. One can see that the local PageRank vectors are much closer to the global PageRank segments than the uniform vectors are. So a concatenation of the local PageRank vectors may form a better start vector for the standard PageRank iteration than the uniform vector.

The relative ordering of pages *within* a host induced by local PageRank scores is generally close to the intra-host ordering induced by the global PageRank scores. To compare the orderings, we measure the average "distance" betwen the local PageRank vectors $\vec{l}_J$ and global PageRank segments $\vec{g}_J$. The KDist distance measure, based on Kendall's-τ rank correlation and used for comparing induced rank orders, is defined as follows:

Consider two partially ordered lists of URLs, τ_1 and τ_2, each of length m. Let U be the union of the URLs in τ_1 and τ_2. If δ_1 is $U - \tau_1$, then let τ_1' be the extension of τ_1, where τ_1' contains δ_1 appearing after all the URLs in τ_1.[3] We extend τ_2 analogously to yield τ_2'. KDist is then defined as:

$$\mathrm{KDist}(\tau_1, \tau_2) = \frac{|\{(u,v) : \tau_1', \tau_2' \text{ disagree on order of } (u,v), u \neq v\}|}{(|U|)(|U|-1)}$$

In other words, $\mathrm{KDist}(\tau_1, \tau_2)$ is the probability that τ_1' and τ_2' disagree on the relative ordering of a randomly selected pair of distinct nodes $(u,v) \in U \times U$. In the current work, we only compare lists containing the same sets of elements, so that KDist is identical to Kendall's τ distance.

[3] The URLs in δ are placed with the *same* ordinal rank at the end of τ.

Web Page	Local	Global
http://aa.stanford.edu	0.2196	0.4137
http://aa.stanford.edu/aeroastro/AAfolks.html	0.0910	0.0730
http://aa.stanford.edu/aeroastro/AssistantsAero.html	0.0105	0.0048
http://aa.stanford.edu/aeroastro/EngineerAero.html	0.0081	0.0044
http://aa.stanford.edu/aeroastro/Faculty.html	0.0459	0.0491
http://aa.stanford.edu/aeroastro/FellowsAero.html	0.0081	0.0044
http://aa.stanford.edu/aeroastro/GraduateGuide.html	0.1244	0.0875
http://aa.stanford.edu/aeroastro/Labs.html	0.0387	0.0454
http://aa.stanford.edu/aeroastro/Links.html	0.0926	0.0749
http://aa.stanford.edu/aeroastro/MSAero.html	0.0081	0.0044
http://aa.stanford.edu/aeroastro/News.html	0.0939	0.0744
http://aa.stanford.edu/aeroastro/PhdAero.html	0.0081	0.0044
http://aa.stanford.edu/aeroastro/aacourseinfo.html	0.0111	0.0039
http://aa.stanford.edu/aeroastro/aafaculty.html	0.0524	0.0275
http://aa.stanford.edu/aeroastro/aalabs.html	0.0524	0.0278
http://aa.stanford.edu/aeroastro/admitinfo.html	0.0110	0.0057
http://aa.stanford.edu/aeroastro/courseinfo.html	0.0812	0.0713
http://aa.stanford.edu/aeroastro/draftcourses.html	0.0012	0.0003
http://aa.stanford.edu/aeroastro/labs.html	0.0081	0.0044
http://aa.stanford.edu/aeroastro/prospective.html	0.0100	0.0063
http://aa.stanford.edu/aeroastro/resources.html	0.0112	0.0058
http://aa.stanford.edu/aeroastro/visitday.html	0.0123	0.0068

Table 4: The local PageRank vector $\vec{l}_J$ for the domain aa.stanford.edu (left) compared to the global PageRank segment $\vec{g}_J$ corresponding to the same pages. The local PageRank vector has a similar ordering to the normalized components of the global PageRank vector. The discrepancy in actual ranks is largely due to the fact that the local PageRank vector does not give enough weight to the root node http://aa.stanford.edu.

The average distance $\mathrm{KDist}(\vec{l}_J, \vec{g}_J)$ is 0.0571 for the hosts in STANFORD/BERKELEY. Notice that this distance is low. This observation means that the ordering induced by the local PageRank is close to being correct, and thus suggests that the majority of the L_1 error in the comparison of local and global PageRanks comes from the miscalibration of a few pages on each host. Indeed the miscalibration may be among important pages; as we discuss next, this miscalibration is corrected by the final step of our algorithm. Furthermore, the relative rankings of pages on *different* hosts is unkown at this point. For these reasons, we do not suggest using local PageRank for ranking pages; we use it only as a tool for computing the *global* PageRank more efficiently.

Table 4 confirms this observation for the host aa.stanford.edu. Notice that the ordering is preserved, and a large part of the discrepancy is due to http://aa.stanford.edu. The local PageRank computation gives too little weight to the root node. Since the elements of the local vector sum to 1, the ranks of all of the other pages are upweighted.

It should be noted that errors of this pattern (where the majority of L_1 error comes from the miscalibration of a few pages) are fixed easily, since once these miscalibrated pages are fixed (by, for example, a few iterations of global PageRank), the rest of the pages fall into place. Errors that are more random take longer to fix. We observe this empirically, but do not include these experiments here for space considerations.

This suggests a stopping criteria for local PageRank computations. At each stage in a local PageRank computation, one could compute the Kendall's-τ residual (the Kendall's-τ distance between the current iteration $\vec{l}_J^{(k)}$ and the previous iteration $\vec{l}_J^{(k-1)}$). When the Kendall's-τ resid-

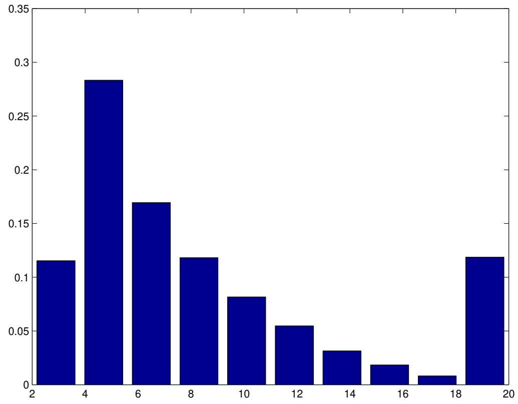


Figure 4: Local PageRank convergence rates for hosts in DNR-LARGEWEB. The x-axis is the number of iterations until convergence to a tolerance of 10^{-1}, and the y-axis is the fraction of hosts that converge in a given number of iterations.

ual is 0, that means the ordering is correct, and the local PageRank computation can be stopped.

4.3.2 Local PageRank Convergence Rates

Another interesting question to investigate is how quickly the local PageRank scores converge. In Figure 4, we show a histogram of the number of iterations it takes for the local PageRank scores for each host in DNR-LARGEWEB to converge to an L_1 residual $< 10^{-1}$. Notice that most hosts converge to this residual in less than 12 iterations.

Interestingly, there is no correlation between the convergence rate of a host and the host's size. Rather, the convergence rate is primarily dependent on the extent of the nested block structure within the host. That is, hosts with strong nested blocks are likely to converge slowly, since they represent slow-mixing Markov chains. Hosts with a more random connection pattern converge faster, since they represent a fast-mixing Markov chain.

This observation suggests that one could make the local PageRank computations even faster by wisely choosing the blocks. That is, if a host has a strong nested block structure, use the directories within that host as your blocks. However, this is not a crucial issue, because we show in Section 5 that the local PageRank computations can be performed in a distributed fashion in parallel with the crawl. Therefore, reducing the cost of the local PageRank computations are not a bottleneck for computing PageRank with our scheme, as the local computations can be pipelined with the crawl.[4]

4.4 Estimating the Relative Importance of Each Block

In this section, we describe computing the relative importance, or *BlockRank*, of each block. Assume there are k blocks in the web. To compute BlockRanks, we first create the *block graph* B, where each vertex in the graph corresponds to a block in the web graph. An edge between two pages in the web is represented as an edge between the corresponding blocks (or a self-edge, if both pages are in the same block). The edge weights are determined as follows: the weight of an edge between blocks I and J is defined to be the sum of the edge-weights from pages in I to pages in J in the web graph, weighted by the local PageRanks of the linking pages in block I.

That is, if A is the web graph and l_i is the local PageRank of page i in block I, then weight of edge B_{IJ} is given by:

$$B_{IJ} = \sum_{i \in I, j \in J} A_{ij} \cdot l_i$$

We can write this in matrix notation as follows. Define the local PageRank matrix L to be the $n \times k$ matrix whose columns are the local PageRank vectors $\vec{l}_J$.

$$L = \begin{pmatrix} \vec{l}_1 & \vec{0} & \cdots & \vec{0} \\ \vec{0} & \vec{l}_2 & \cdots & \vec{0} \\ \vdots & \vdots & \ddots & \vdots \\ \vec{0} & \vec{0} & \cdots & \vec{l}_K \end{pmatrix}$$

Define the matrix S to be the $n \times k$ matrix that has the same structure as L, but whose nonzero entries are all replaced by 1. Then the block matrix B is the $k \times k$ matrix:

$$B = L^T A S$$

Notice that B is a transition matrix where the element B_{IJ} represents the transition probability of block I to block J. That is:

$$B_{IJ} = p(J|I)$$

Once we have the $k \times k$ transition matrix B, we may use the standard PageRank algorithm on this reduced matrix to compute the BlockRank vector $\vec{b}$. That is:

$$\vec{b} = \textit{pageRank}(B, \vec{v}_k, \vec{v}_k)$$

where $\vec{v}_k$ is the uniform k-vector $[\frac{1}{k}]_{k \times 1}$.

Note that this is the same as computing the stationary distribution of the transition matrix $c \cdot B + (1-c)E_k$, where we define $E_k = [1]_{k \times 1} \times \vec{v_k}^T$. The analogy to the random surfer model of [14] is this: we imagine a random surfer going from block to block according to the transition probability matrix B. At each stage, the surfer will get bored with probability $1 - c$ and jump to a different *block*.

4.5 Approximating Global PageRank using Local PageRank and BlockRank

In this section, we find an estimate to the global PageRank vector $\vec{x}$. At this point, we have two sets of rankings. Within each block J, we have the local PageRanks $\vec{l}_J$ of the pages in the block. We also have the BlockRank vector $\vec{b}$ whose elements b_J are the BlockRank for each block J, measuring the relative importance of the blocks. We may now approximate the global PageRank of a page $j \in J$ as

[4] Generally this requires a site-based crawler (such as the WebBase crawler [4]) which maintains a pool of active hosts, and crawls hosts to completion before adding new hosts to the pool.

its local PageRank l_j, weighted by the BlockRank b_J of the block in which it resides. That is,

$$x_j^{(0)} = l_j \cdot b_J$$

In matrix notation, this is:

$$\vec{x}^{(0)} = L\vec{b}$$

Recall that the local PageRanks of each block sum to 1. Also, the BlockRanks sum to 1. Therefore, our approximate global PageRanks will also sum to 1. The reasoning follows: the sum of of our approximate global PageRanks $sum(x_j) = \sum_j x_j$ can be written as a sum over blocks

$$sum(x_j) = \sum_J \sum_{j \in J} x_j$$

Using our definition for x_j from Equation 4.5

$$sum(x_j) = \sum_J \sum_{j \in J} l_j b_J = \sum_J b_J \sum_{j \in J} l_J$$

Since the local PageRanks for each domain sum to 1 ($\sum_{j \in J} l_j = 1$)

$$sum(x_j) = \sum_J b_J$$

And since the BlockRanks also sum to 1 ($\sum_J b_J = 1$)

$$sum(x_j) = 1$$

Therefore, we may use our approximate global PageRank vector $\vec{x}^{(0)}$ as a start vector for the standard PageRank algorithm.

4.6 Using This Estimate as a Start Vector

In order to compute the true global PageRank vector $\vec{x}$ from our approximate PageRank vector $\vec{x}^{(0)}$, we simply use it as a start vector for standard PageRank. That is:

$$\vec{x} = pageRank(G, \vec{x}^{(0)}, \vec{v})$$

where G is the graph of the web, and $\vec{v}$ is the uniform distribution over root nodes. In Section 7, we show how to compute different personalizations quickly once $\vec{x}$ has been computed. The BlockRank algorithm for computing PageRank, presented in the preceding sections, is summarized by Algorithm 3, given in the appendix.

5 Advantages of BlockRank

The BlockRank algorithm has four major advantages over the standard PageRank algorithm.

Advantage 1 A major speedup of our algorithm comes from caching effects. All of the host-blocks in our crawl are small enough so that each block graph fits in main memory, and the vector of ranks for the active block largely fits in the CPU cache. As the full graph does not fit entirely in main memory, the local PageRank iterations thus require less disk i/o then the global computations. The full rank vectors do fit in main memory; however, using the sorted link structure[5] dramatically improves the memory access patterns to the rank vector. Indeed, if we use the sorted link structure, designed for BlockRank, as the input instead to the *standard* PageRank algorithm, the enhanced locality of reference to the rank vectors cuts the time needed for each iteration of the standard algorithm by over 1/2: from 6.5 minutes to 3.1 minutes for each iteration on DNR-LARGEWEB!

Advantage 2 In our BlockRank algorithm, the local PageRank vectors for many blocks will converge quickly; thus the computations of those blocks may be terminated after only a few iterations. This increases the effectiveness of the local PageRank computation by allowing it to expend more computation on slowly converging blocks, and less computation on faster converging blocks. Note for instance in Figure 4 that there is a wide range of rates of convergence for the blocks. In the standard PageRank algorithm, iterations operate on the whole graph; thus the convergence bottleneck is largely due to the slowest blocks. Much computation is wasted recomputing the PageRank of blocks whose local computation has already converged.

Advantage 3 The local PageRank computations in Step 1 of the BlockRank algorithm can be computed in a completely parallel or distributed fashion. That is, the local PageRanks for each block can be computed on a separate processor, or computer. The only communication required is that, at the end of Step 1, each computer should send their local PageRank vector $\vec{l_j}$ to a central computer that will compute the global PageRank vector. If our graph consists of n total pages, the net communication cost consists of $8n$ bytes (if using 8-byte double precision floating point values). Naive parallelization of the computation that does not exploit block structure would require a transfer of $8n$ bytes *after each iteration*, a significant penalty. Furthermore, the local PageRank computations can be pipelined with the web crawl. That is, the local PageRank computation for a host can begin as a separate process as soon as the crawler finishes crawling the host. In this case, only the costs of Steps 2–4 of the BlockRank algorithm become rate-limiting.

Advantage 4 In several scenarios, the local PageRank computations (e.g., the results of Step 1) can be reused during future applications of the BlockRank algorithm. Consider for instance news sites such as cnn.com that are crawled more frequently then the general web. In this case, after a crawl of cnn.com, if we wish to recompute the global PageRank vector, we can rerun the BlockRank algorithm, except that in Step 1 of our algorithm, only the local PageRanks for the cnn.com block need to be recomputed. The remaining local PageRanks will be unchanged, and can be reused in Steps 2–3. In this way, we can also reuse

[5] As in Section 3, this entails assigning document ids in lexicographic order of the url (with the components of the full hostname reversed).

Step	Wallclock time
1	17m 11s
2	7m 40s
3	0m 4s
4	56m 24s
Total	**81m 19s**

Table 5: Running times for the individual steps of BlockRank for $c = 0.85$ in achieving a final residual of $< 10^{-3}$.

Algorithm	Wallclock time
Standard	180m 36s
Standard (using url-sorted links)	87m 44s
BlockRank (no pipelining)	81m 19s
BlockRank (w/ pipelining)	57m 06s

Table 6: Wallclock running times for 4 algorithms for computing PageRank with $c = 0.85$ to a residual of less than 10^{-3}.

	PageRank	BlockRank
STANFORD/BERKELEY	50	27
LARGEWEB	28	18

Table 7: Number of iterations needed to converge for standard PageRank and for BlockRank (to a tolerance of 10^{-4} for STANFORD/BERKELEY, and 10^{-3} for LARGEWEB).

the local PageRank computations for the case of computing several "personalized" PageRank vectors. We further discuss personalized PageRank in Section 7, and graph updates in Section 8.

6 Experimental Results

In this section, we investigate the speedup of BlockRank compared to the standard algorithm for computing PageRank. The speedup of our algorithm for typical scenarios comes from the first three advantages listed in Section 5. The speedups are due to less expensive iterations, as well as fewer total iterations. (Advantage 4 is discussed in subsequent sections)

We begin with the scenario in which PageRank is computed after the completion of the crawl; we assume that only Step 0 of the BlockRank algorithm is computed concurrently with the crawl. As mentioned in Advantage 1 from the previous section, simply the improved reference locality due to blockiness, exposed by lexicographically sorting the link matrix, achieves a speedup of a factor of 2 in the time needed for each iteration of the standard PageRank algorithm. This speedup is completely independent of the value chosen for c, and does not affect the rate of convergence as measured in number of iterations required to reach a particular L_1 residual.

If instead of the standard PageRank algorithm, we use the BlockRank algorithm on the block structured matrix, we gain the full benefit of Advantages 1 and 2; the blocks each fit in main memory, and many blocks converge more quickly than the convergence of the entire web. We compare the wallclock time it takes to compute PageRank using the BlockRank algorithm in this scenario, where local PageRank vectors are computed serially after the crawl is complete, with the wallclock time it takes to compute PageRank using the standard algorithm given in [14]. Table 5 gives the running times of the 4 steps of the BlockRank algorithm on the LARGEWEB dataset. The first 3 rows of Table 6 give the wallclock running times for standard PageRank, standard PageRank using the url-sorted link matrix, and the full BlockRank algorithm computed after the crawl. We see there is a small additional speedup for BlockRank on top of the previously described speedup. Subsequently, we will describe a scenario in which the costs of Steps 1–3 become largely irrelevant, leading to further effective speedups.

In this next scenario, we assume that the cost of Step 1 can be made negligible in one of two ways: the local PageRank vectors can be pipelined with the web crawl, or they can be computed in parallel after the crawl. If the local PageRank vectors are computed as soon as possible (e.g., as soon as a host has been fully crawled), the majority of local PageRank vectors will have been computed by the time the crawl is finished. Similarly, if the local PageRank vectors are computed after the crawl, but in a distributed manner, using multiple processors (or machines) to compute the PageRank vectors independently, the time it takes to compute the local PageRanks will be low compared to the standard PageRank computation. Thus, only the running time of Steps 2–4 of BlockRank will be relevant in computing net speedup. The contruction of B is the dominant cost of Step 2, but this too can be pipelined; Step 3 has negligible cost. Thus the speedup of BlockRank in this scenario is determined by the increased rate of convergence in Step 4 that comes from using the BlockRank approximation $\vec{x}^{(0)}$ as the start vector. We now take a closer look at the relative rates of convergence. In Figure 5(a), we show the convergence rate of standard PageRank, compared to the convergence of Step 4 of BlockRank on the STANFORD/BERKELEY dataset for a random jump probability $1 - c = 0.15$ (i.e., $c = 0.85$). Note that to achieve convergence to a residual of 10^{-4}, using the BlockRank start vector leads to a speedup of a factor of 2 on the STANFORD/BERKELEY dataset. The LARGEWEB dataset yielded an increase in convergence rate of 1.55. These results are summarized in Table 7. Combined with the first effect described above (from the sorted link structure), in this scenario, our algorithm yields a net speedup of over 3. (For higher values of c, as explored in [11], the speedup is even more significant; for example we got a 10-times speedup on the STANFORD/BERKELEY dataset when we set $c = 0.99$.)

These results are the most significant speedup results to date for computing PageRank, and the only results showing significant speedup for $c = 0.85$ on large datasets. Also, it should be noted that the BlockRank algorithm can be used in conjunction with other methods of accelerating PageRank, such as Quadratic Extrapolation [11], or Gauss-Seidel [6, 1] (or both). These methods would simply be applied to Step 4 in the BlockRank algorithm. When used in conjunction with these methods, one should expect even

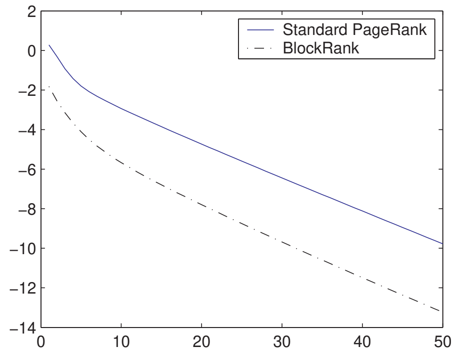


Figure 5: Convergence rates for standard PageRank (solid line) vs. BlockRank (dotted line). The x-axis is the number of iterations, and the y-axis is the log of the L_1-residual. STANFORD/BERKELEY data set; $c = 0.85$.

faster convergence for BlockRank; these hybrid approaches are left for future study.

7 Personalized PageRank

In [14], it was originally suggested that, by changing the random jump vector $\vec{v}$ to be nonuniform, the resultant PageRank vector can be biased to prefer certain kinds of pages. For example, a random surfer interested in sports may get bored every once in a while and jump to http://www.espn.com, while a random surfer interested in current events may instead jump to http://www.cnn.com when bored. While personalized PageRank is a compelling idea, in general it requires computing a large number of PageRank vectors.

We use the BlockRank algorithm and a simple restriction on the jump behavior of the random surfer to dramatically reduce the computation time of personalized PageRank. The restriction is this: instead of being able to choose a distribution over *pages* to which he jumps when he's bored, the random surfer may choose *hosts*. For example, the random surfer interested in sports may jump to the www.espn.com host, but he may not, for example, jump to http://www.espn.com/ncb/columns/forde_pat/index.html. We can then encode the personalization vector in the k-dimensional vector $\vec{v}_k$ (where k is the number of host-blocks in the web) that is a distribution over different hosts.

With this restriction, the local PageRank vectors $\vec{l}_J$ will not change for different personalizations. In fact, since the local PageRank vectors $\vec{l}_J$ do not change for different personalizations, neither does the block matrix B.

Only the BlockRank vector $\vec{b}$ will change for different personalizations. Therefore, we only need to recompute the BlockRank vector $\vec{b}$ for each block-personalization vector $\vec{v}_k$.

The personalized PageRank computation could proceed as follows. Assuming you have already computed a generic PageRank vector once using the BlockRank algorithm, and have stored the block-transition matrix B, the personalized BlockRank algorithm is simply the last 3 steps of the generic BlockRank algorithm.

7.1 Inducing Random Jump Probabilities Over Pages

The Personalized BlockRank algorithm requires that the random surfer not have the option of jumping to a specific page when he bores (he may only jump to the host). However, the last step in the BlockRank algorithm requires a random jump probability distribution $\vec{v}$ over *pages*. Thus, we need to induce the probability $p(j)$ that the random surfer will jump to a page j if we know the probability $p(J)$ that he will jump to host J in which page j resides. We induce this probability as follows:

$$p(j) = p(J)p(j|J) \tag{2}$$

That is, the probability that the random surfer jumps to page j is the probability that he will jump to host J, times the probability of being at page j given that he is in host J.

Since the local PageRank vector $\vec{l}_J$ is the stationary probability distribution of pages within host J, $p(j|J)$ is given by the element of $\vec{l}_J$ corresponding to page j. Therefore, the elements L_{jJ} of the matrix L correspond to $L_{jJ} = p(j|J)$. Also, by definition, the elements $(v_k)_J = p(J)$. Therefore, in matrix notation, Equation 2 can be written as $\vec{v} = L\vec{v}_k$.

7.2 Using "Better" Local PageRanks

If we have already computed the generic PageRank vector $\vec{x}$, we have even "better" local PageRank vectors than we began with. That is, we can normalize segments of $\vec{x}$ to form the normalized global PageRank segments $\vec{g}_J$ as described in Section 4. These scores are of course better estimates of the relative magnitudes of pages within the block than the local PageRank vectors $\vec{l}_J$, since they are derived from the generic PageRank vector for the full web. So we can modify Personalized BlockRank as follows. Let us define the matrix H in a manner similar to the way we defined L, except using the normalized global PageRank segments $\vec{g}_J$ rather than the local PageRank vectors $\vec{l}_J$. Again, we only need to compute H once. We define the matrix B_H to be similar to the matrix B as defined in Equation 4.4, but using H instead of L:

$$B_H = H^T AS \tag{3}$$

7.3 Experiments

We test this algorithm by computing the Personalized PageRank of a random surfer who is a graduate student in linguistics at Stanford. When he bores, he has an 80% probability of jumping to the linguistics host www-linguistics.stanford.edu, and a 20% probability of jumping to the main Stanford host www.stanford.edu. Figure 6 shows that the speedup of computing the Personalized PageRank for this surfer shows comparable speedup benefits to standard BlockRank. However, the main benefit is that the local PageRank vectors do not need to be computed at all for Personalized BlockRank. The matrix H is formed from

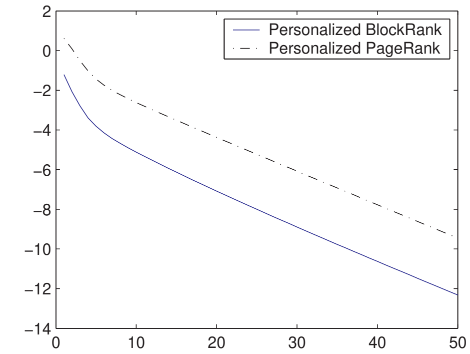


Figure 6: Convergence of Personalized PageRank computations using standard PageRank and Personalized BlockRank.

the already computed generic PageRank vector. Therefore, the overhead to computing Personalized PageRank vectors using the Personalized BlockRank algorithm is minimal.

8 Node Updates

We can also utilize the strategy of reusing Local PageRank vectors when we wish to recompute PageRank after several pages have been added or removed from the web. Since the web is highly dynamic, with web pages being added or removed all the time, this is an important problem to address. In particular, we wish to crawl certain hosts, such as daily news providers such as cnn.com more frequently than others.

If we use BlockRank to compute the PageRank vector $\vec{x}$, and store the local PageRank vectors $\vec{l}_J$, then we only need to recompute the local PageRanks of those hosts to which pages have been added or removed at each update.

9 Conclusion

We have shown that the hyperlink graph of the web has a nested block structure, something that has not yet been thoroughly investigated in studies of the web. We exploit this structure to compute PageRank in a fast manner using an algorithm we call BlockRank. We show empirically that BlockRank speeds up PageRank computations by factors of 2 and higher, depending on the particular scenario. There are a number of areas for future work: finding the "best" blocks for BlockRank by splitting up what would be slow-mixing blocks with internal nested block structure; using the block structure for hyperlink-based algorithms other than web search, such as in clustering or classification; and exploring more fully the topics of of updates and personalized PageRank.

10 Acknowledgments

This paper is based on work supported in part by the National Science Foundation under Grant No. IIS-0085896 and Grant No. CCR-9971010, and in part by the Research Collaboration between NTT Communication Science Laboratories, Nippon Telegraph and Telephone Corporation and CSLI, Stanford University (research project on Concept Bases for Lexical Acquisition and Intelligently Reasoning with Meaning).

References

[1] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin. PageRank computation and the structure of the web: Experiments and algorithms. In *Proceedings of the Eleventh International World Wide Web Conference, Poster Track*, 2002.

[2] K. Bharat, B.-W. Chang, M. Henzinger, and M. Ruhl. Who links to whom: Mining linkage between web sites. In *Proceedings of the IEEE International Conference on Data Mining*, November 2001.

[3] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the Ninth International World Wide Web Conference*, 2000.

[4] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proceedings of the Eleventh International World Wide Web Conference*, 2002.

[5] P.-J. Courtois. *Queueing and Computer System Applications*. Academic Press, 1977.

[6] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1996.

[7] G. Grimmett and D. Stirzaker. *Probability and Random Processes*. Oxford University Press, 1989.

[8] T. H. Haveliwala. Topic-sensitive PageRank. In *Proceedings of the Eleventh International World Wide Web Conference*, 2002.

[9] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. Webbase: A repository of web pages. In *Proceedings of the Ninth International World Wide Web Conference*, 2000.

[10] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.

[11] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating PageRank computations. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.

[12] J. Kleinberg, S. R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web as a graph: Measurements, models, and methods. In *Proceedings of the International Conference on Combinatorics and Computing*, 1999.

[13] D. McAllister, G. Stewart, and W. Stewart. On a rayleigh-riz refinement technique for nearly uncoupled stochastic matrices. *Linear Algebra and Its Applications*, 60:1–25, 1984.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. *Stanford Digital Libraries Working Paper*, 1998.

[15] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *Proceedings of the IEEE Intl. Conference on Data Engineering*, March 2003.

[16] M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in pagerank. In *Advances in Neural Information Processing Systems*, volume 14. MIT Press, Cambridge, MA, 2002.

[17] H. A. Simon and A. Ando. Aggregation of variables in dynamic systems. *Econometrica*, 29:111–138, 1961.

Appendix

This appendix summarizes the BlockRank and Personalized BlockRank algorithms presented in this paper. The standard BlockRank algorithm for computing PageRank is summarized in Algorithm 3.[6]

0. Sort the web graph lexicographically as described in Section 3, exposing the nested block structure of the web.
1. Compute the local PageRank vector $\vec{l}_J$ for each block J.

foreach *block* J **do**
$\vec{l}_J = \textit{pageRank}(G_{JJ}, \vec{s}_J, \vec{v}_J);$
end

2. Compute block transition matrix B and BlockRanks $\vec{b}$.

$$B = L^T AS$$
$$\vec{b} = \textit{pageRank}(B, \vec{v}_k, \vec{v}_k)$$

3. Find an approximation $\vec{x}^{(0)}$ to the global PageRank vector $\vec{x}$ by weighting the local PageRanks of pages in block J by the BlockRank of J.

$$\vec{x}^{(0)} = L\vec{b}$$

4. Use this approximation as a start vector for a standard PageRank iteration.

$$\vec{x}^{(0)} = pageRank(G, \vec{x}^{(0)}, \vec{v})$$

Algorithm 3: BlockRank Algorithm

The Personalized BlockRank algorithm for computing different personalizations of PageRank is summarized in Algorithm 4. Assuming you have already computed a generic PageRank vector once using the BlockRank algorithm, and have computed H from the generic PageRank vector $\vec{x}$, and B_H as defined in Equation 3, the algorithm proceeds as follows:

For a given block-personalization vector $\vec{v}_k$,
1. Compute the personalized BlockRank vector $\vec{b}$

$\vec{b} = \textit{pageRank}(B_H, \vec{s}, \vec{v}_k)$, where $\vec{s}$ is a uniform start vector.

2. Find an approximation $\vec{x}^{(0)}$ to the global PageRank vector $\vec{x}$ by weighting the local PageRanks of pages in block J by the personalized BlockRank of J.

$$\vec{x}^{(0)} = H\vec{b}$$

3. Induce the personalization vector $\vec{v}$ over pages from the personalization vector over hosts $\vec{v}_k$.

$$\vec{v} = H\vec{v}_k$$

4. Use this approximation as a start vector for a standard PageRank iteration.

$$\vec{x} = pageRank(G, \vec{x}^{(0)}, \vec{v})$$

Algorithm 4: Personalized BlockRank Algorithm

[6] In the newest iteration of the WebBase system, the site-based crawler stores pages from different hosts in different files, making Step 0 practical. Furthermore, the WebBase system currently maintains a sorted list of the urls for efficiently encoding the url-lookup table. For these reasons, we do not include the cost of Step 0 in future discussion unless explicitly stated.