A Uniform Approach to Accelerated PageRank Computation

Frank McSherry
Microsoft Research, SVC
1065 La Avenida
Mountain View, CA, USA 94043
mcsherry@microsoft.com

ABSTRACT

In this note we consider a simple reformulation of the traditional power iteration algorithm for computing the stationary distribution of a Markov chain. Rather than communicate their current probability values to their neighbors at each step, nodes instead communicate only changes in probability value. This reformulation enables a large degree of flexibility in the manner in which nodes update their values, leading to an array of optimizations and features, including faster convergence, efficient incremental updating, and a robust distributed implementation.

While the spirit of many of these optimizations appear in previous literature, we observe several cases where this unification simplifies previous work, removing technical complications and extending their range of applicability. We implement and measure the performance of several optimizations on a sizable (34M node) web subgraph, seeing significant composite performance gains, especially for the case of incremental recomputation after changes to the web graph.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Experimentation, Performance, Theory

Keywords

PageRank, web graph, link analysis, random walks

1. INTRODUCTION

Motivated largely by the success and scale of Google's PageRank ranking function, much research has emerged on efficiently computing the stationary distributions of web-scale Markov chains, the mathematical mechanism underlying PageRank. The main challenge is that the web graph is so large that its edges typically only exist in external memory and an explicit representation of its stationary distribution just barely fits in to main memory. The time required to compute the stationary distribution is on the order of tens of hours to days, and constant factor improvements in running times can save substantial time and money. Even for the common researcher with interests in ranking research, computing and recomputing vectors of ranks is a time consuming processes that greatly limits research throughput.

As such, much work has been done on accelerating the performance of PowerIteration, the traditional approach to computing stationary distributions. These optimizations cover a spectrum of techniques, ranging from transformations to the Markov chain that accelerate mixing to efficient heuristic updates that behave like PowerIteration to clever reuse of previously computed solutions. Most of these techniques are developed and evaluated in isolation, and it is unclear to what degree they can be effectively combined, both in terms of implementation and performance.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

1.1 Notation and Terminology

Throughout this note we will frequently refer to vectors, matrices, and the scalars they comprise. For clarity, we consistently use lowercase letters (x) for vectors and capital letters (A) for matrices. For each, subscripted quantities (x_u and A_{uv}) are used to reference the scalar values at the indexed coordinates.

1.2 PageRank and PowerIteration

PageRank [2] is a system of scoring nodes in a directed graph based on the stationary distribution of a random walk on the directed graph. Conceptually, the score of a node corresponds to the frequency with which the node is visited as an individual strolls randomly through the graph. For technical reasons, the random walk is also encouraged to occasionally reset to a prespecified distribution, overcoming issues of weakly connected components in which a random surfer might get stuck and accelerating the rate at which a random walk approaches the stationary distribution.

A random walk on n nodes can be described by a $n \times n$ matrix P, where entry P_{vu} is the probability that from node u the walk next arrives at node v. Starting from a distribution x over the nodes (x is a vector of n entries that are non-negative and sum to one), after one step the distribution becomes Px, and more generally after i steps the distribution becomes $P^i x$.

We can decompose P into those transitions due to traversing a web link, and those transitions due to random reseting. Let the sparse matrix A have entries A_{vu} equal to the probability that from node u the walk traverses the link (u, v) to node v. Additionally, we define the vector r with each coor-

dinate $r_u = 1 - \sum_v A_{vu}$, equal to the probability that the that the walk chooses not to follow an arc from node u and instead resets randomly to a node v chosen with probability proportional to d_v. P can then be written as $P = A + dr^T$, capturing both of the types of transitions.

PowerIteration is the traditional manner of computing the stationary distribution of P, explicitly simulating the dissemination of probability mass by repeatedly applying P to a supplied initial distribution x. Under modest assumptions, e. g. that all entries of d and r are positive, for any initial distribution x, $P^i x$ converges to a unique stationary distribution as i increases.

PowerIteration(P, x)

1. While (not converged)
   (a) Set $x = Px$

While P itself is a dense matrix, every node can reset to any other node, we can efficiently compute Px by viewing it as $Ax + dr^T x$. We assume that A is stored on disk in a sparse format, perhaps as a list of $(source, target, value)$ triples, though there are more compact representations. Ax is then computed using sparse matrix-vector multiplication: since $(Ax)_v = \sum_v A_{vu} x_u$, we can populate the result vector by scanning the edge file, for each non-zero A_{vu} adding $A_{vu} x_u$ to coordinate v of the result. We can produce the vector $dr^T x$ by determining $r^T x$ in a pass over r and x, and scaling d appropriately before adding it to Ax.

For acceptable performance, we may only perform sequential access to the edge file, which is too large to fit into main memory. Generally speaking, the number of non-zero entries in A is the limiting performance factor, both because of our need to scan over the edge file to read these entries, and also the random accesses to x each entry requires. The other operations, vector addition, scaling, and inner product, can all be done using sequential access to main memory.

2. AN UPDATE-BASED ALGORITHM

Oddly, we start our generalization of PowerIteration by restricting the problem we address. There are many vectors satisfying $x = Px$; any solution x can be multiplied by an arbitrary scalar value and still satisfy the equality. Typically we focus our attention on finding the vector x with $\|x\|_1 = 1$. Let us instead focus on finding the vector x for which $r^T x = 1$, and for which

$$x \;=\; Px \;=\; Ax + dr^T x \;=\; Ax + d\,.$$

As we will see in Theorem 1, if $x = Ax + d$, then $x = Px$. Normalized, this vector is the stationary distribution of P.

Consider an analog of PowerIteration in which we repeatedly set $x = Ax + d$. As with PowerIteration, this iterative process will converge, and it converges to a vector satisfying $x = Ax + d$. We can monitor convergence through the vector $y = Ax - x + d$: so long as y is non-zero, x has not yet converged. But, y also tells us the direction to update x; we advance to the next iterate of x by adding y, yielding $Ax + d$. This first role is crucial, we must bring y to zero, but we needn't be so rigid as to only ever add y to x. We might instead add other vectors to x that yield forward progress, maintaining y both as a convergence criteria, but also for guidance in choosing updates to x.

Consider an algorithm that monitors $y = Ax - x + d$, but is free to choose an arbitrary update vector z at any step, advancing from x to $x + z$. It is not hard to appropriately update y, as its new value satisfies

$$A(x + z) - (x + z) + d \;=\; y + Az - z\,.$$

For any update vector z, we can update y by passing z through the matrix A, adding the result to y, and subtracting z. Intuitively, z is extracted from y and propagated across the links in A, informing nodes of changes in their parent's values and insisting that they now update in turn.

Operationally, this is exactly the algorithmic framework that we will consider. However, it will be useful not to fix y in terms of a particular A, x, and d, but rather let it be specified as an input parameter, properly determined before the method is invoked.

UpdateIteration (A, x, y)

1. While (updates y remain):
   (a) Choose an update vector z.
   (b) Set $x = x + z$.
   (c) Set $y = y + (Az - z)$.

While this framework is presently little more than a system of bookkeeping, we will solidify how one might choose z to shrink $\|y\|_1$, and which choices lead to efficient algorithms.

We now state two theorems regarding the limit and rate of convergence of UpdateIteration(A, x, y). The proofs, while short, are rote and unilluminating, and are defered to Appendix A. We first argue that choosing $y = Ax - x + d$ leads to a stationary vector of $P = A + dr^T$, but also, in a rather oblique manner, describe where x ends up if we start UpdateIteration(A, x, y) with an arbitrary y.

THEOREM 1. *For vectors x, y, d and substochastic matrix A, if $y = Ax - x + d$ and d is a non-negative vector, then defining the stochastic matrix $P = A + dr^T / \|d\|_1$,*

$$\|Px - x\|_1 \;\leq\; 2\|y\|_1 \quad \textit{and} \quad \|x\|_1 \;\geq\; \|d\|_1 - \|y\|_1\,.$$

To reiterate, Theorem 1 not only describes the correct initial value of y to arrive at a stationary vector of $P = A + dr^T$, but also says that for any A, x, y, if the vector d satisfying $y = Ax - x + d$ is non-negative, then x arrives as the stationary distribution of a random walk on A that resets to a distribution proportional to d.

While the limit of x is well defined, choosing z arbitrarily clearly need not result in rapid, or any, convergence to this limit. Much as y leads x to its limit, vectors z whose coordinates agree with those of y also exhibit brisk convergence of $\|y\|_1$ to zero.

THEOREM 2. *If each z_u lies between zero and y_u, then*

$$\|y + Az - z\|_1 \quad\leq\quad \|y\|_1 - \sum_u r_u |z_u|\,.$$

When all r_u are equal, the exponential convergence of PowerIteration is a special case of Theorem 2: processing $z = y$ each round reduces $\|y\|_1$ by a factor of $1 - r_u$. Moreover, when r is not uniform Theorem 2 gives a tighter characterization of progress than eigenvalue bounds, which are generally in terms of the smallest r_u value. Finally, and critically, Theorem 2 describes progress made when we process an update $z \neq y$, and informs us as to where in y the progress is being made.

3. ACCELERATION TECHNIQUES

In this section we consider several manners of choosing the vector z in UpdateIteration that give rise to various acceleration techniques. Most have occurred in some form previously in the literature, and we will discuss the often significant differences between their current and previous incarnations. In each case we will find shortcomings of previous techniques that are resolved by casting them in our common framework. Additionally, a significant advantage is the simple manner in which the techniques now compose, both from an algorithmic and performance perspective.

We also present experimental data detailing the performance of the acceleration techniques and compositions we discuss on a 34M page crawl from 2002 containing roughly 800M edges. The pages are organized first by host, where hosts are sorted by crawl discovery order, and within each host by crawl discovery order. Several benefits of such an ordering are discussed in Kamvar et al. [6], who use a more thorough sorting within each host. We choose to order pages by hosts, independent of the significant performance gains noted in [6], because one of our optimizations relies on it, and we require a consistent experimental framework.

For each approach, we plot the total error $\|Px-x\|_1/\|x\|_1$ against computational effort, measured in units of 800M edges processed, corresponding to the effort required by a single pass of PowerIteration. The normalization by $\|x\|_1$ is required because our vector x need not remain at unit norm, and it would be unfair for a vector to achieve small $\|Px-x\|_1$ simply by virtue of a small x. In reading the graphs, the acceleration can be seen by in the ratios of effort along a fixed (horizontal) level of error.

3.1 Sequential Updates

In choosing an update vector z, each coordinate makes a commitment to the update z_u it intends, at which point each update is applied to x and propagated through A in parallel. However, in most implementations these updates will be processed serially, typically reading and propagating each z_u in turn. As the z_u may be chosen arbitrarily, there is no need for a node to commit to a particular value until it is needed. Rather, we can delay the choice of z_u until it is needed, conceptually processing a long series of single coordinate updates of the form $z = (0, \ldots 0, z_u, 0, \ldots 0)$.

Sequential updating allows us to base z_u on a value of y_u that reflects all updates applied thus far, even those applied in the current iteration, allowing us to propagate the effect of a single update multiple times in a single pass over the edge file. Even if the nodes are ordered randomly, roughly half of the edges will point forward in the node order. Updates passed along these edges will be processed again before we complete a pass over the edge file. Well organized graphs can benefit even more, with updates pushed along entire acyclic subgraphs in one pass.

Sequential updating is based on a specific ordering of nodes, and clearly some orderings are better than others. The ordering we use is based on crawl order, which has the peculiar property that 80% of the edge point backwards; crawling very quickly discovers pages with many incoming links, and placing them early reverses the direction of the bulk of their links. As any ordering can easily be run in both orders, forwards or backwards or both, we will also consider the sequential updating in reverse order. Experimentally, alternating direction, forwards then backwards, performs more poorly than either unidirectional approach. This is peculiar, and merits further investigation.

Figures 1 and 2 compare traditional PowerIteration (PI) against sequential updating (SU) and sequential updating applied in reverse order (R-SU). It should be stressed that these techniques exhibit exactly the same data access patterns as traditional PowerIteration, passing linearly over the edge file and probing u in main memory for each edge A_{uv}. The approaches differ only in what they do for each A_{uv} (and the direction of scan, for R-SU). Their running times are effectively identical to PowerIteration.

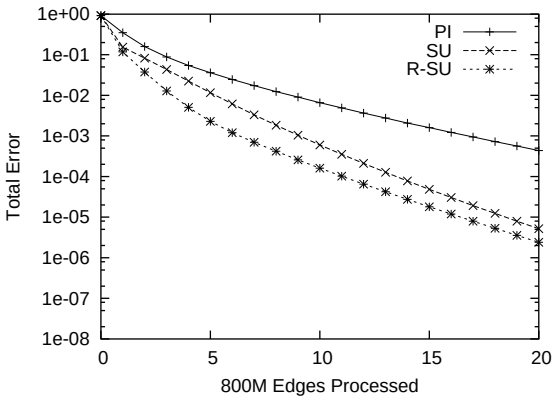

Figure 1: Sequential Updates: Total Error

We see acceleration of nearly 2x and 3x for sequential updates on the crawl ordered and reverse crawl ordered graphs, respectively, with the gap between SU and R-SU diminishing with time.

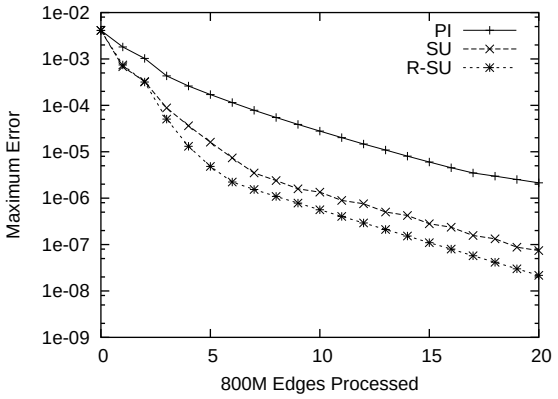

Figure 2: Sequential Updates: Maximum Error

Here we see again a lead of reverse crawl ordering, but the lead is less initially, growing after several iterations.

Related Work: Sequential updating is similar to the Gauss-Seidel approach described by Arasu et al. [1], in which one sequentially sets $x_u = \sum_v P_{uv}x_v$, using the most current values of x_v rather than those of the previous iteration. In contrast with UpdateIteration, the Gauss-Seidel approach requires the graph's edges to be grouped by destination, rather than source, which can substantially complicate data maintenance.

3.2 Reiterated Updates

A large fraction of links in the web graph are *intra-host*, and as such it is common to group pages by host for locality benefits, discussed in [6]. Given such a grouping, after processing the nodes associated with a host, a large fraction of the propagated update z will return to nodes on that host. While this may seem frustrating at first, recall that Theorem 2 says that $\|y\|_1$ decreases by at least $r_u|z_u|$, independent of where Az ends up. Moreover, various caches will retain the data used to process this group, making *reprocessing* it very efficient. Rather than process the next group, reading sequential edge data from disk and probing y_v in main memory, we can reprocess the current group, reading sequential edge data from main memory and probing y_v in the L3 cache. The latter is substantially faster than the former, and represents a good payoff so long as substantial updates remain in the group. As the intra-host edge density is high, we might perform several iterations on a group before its y updates dissipate to other groups.

Another popular grouping is by strongly connected component. Ordered topologically, there is no reason to advance from a component until it has satisfactorily converged, as there are no edges along which updates from subsequent components may return. This approach has the decided advantage that the working set of edges and nodes at any point in time is only as large as the associated strongly connected component, each of which is visited only once. The main disadvantage is that computing strongly connected components is difficult in external memory, and an approximation should probably be used instead. Notice that we do not actually require that the grouping have no back edges, but the fewer that exist, the fewer updates return upstream and the more effective each pass is. Strongly connected components ensure that one pass suffices, but groupings that simply have low reverse edge density are highly effective as well.

There are other interesting groupings that one can imagine (we will discuss some more in Sections 3.3 and 3.4) and the question quickly emerges of which one should be used. In fact, we can use several. Our main constraint is that we access the edge file sequentially, and therefore we must collocate edges from nodes in the same group. If we have the disk space to maintain multiple edge files, we can produce an edge file for each grouping, and choose to use a particular edge file based on our needs at the time. In reading the edge file, we process the collocated nodes in a group, and can easily make multiple passes over this data without tripping over the intervening edges in the original edge file. This approach does not give us locality of reference in the y vector, as we have not actually changed node indices.

Figures 3 and 4 examine the performance benefits of grouping by host and processing each one, two, and three times. We also examine 10x reiteration, though only to demonstrate its limit. As we count operations instead of measure execution time, we will need to make some assumptions about the execution time of subsequent passes. For presentation reasons, we will assume that subsequent iterations are free, which is clearly false. However, the 2x and 3x reiterations result in acceleration of nearly 2x and 3x, respectively, so acceleration clearly exists for more pessimistic assumptions. Actual runtimes suggest that subsequent iterations are cheap, with 2x and 3x reiteration taking roughly 1.25 and 1.50 times as long, respectively, in a not especially well controlled environment. Also, we only need to process the intra-group edges while reiterating, propagating updates along inter-group edges only once finished.

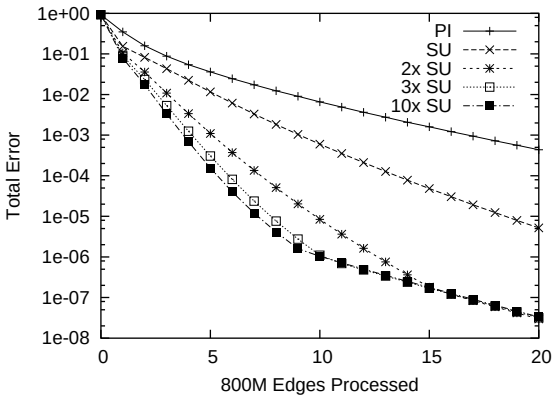

Figure 3: Reiteration: Total Error

The acceleration for total error is almost 2x and 3x over SU, suggesting that reiterated updates can be nearly as effective as multiple passes over the matrix. Of course, there are diminishing returns, visible as 3x and 10x converge.

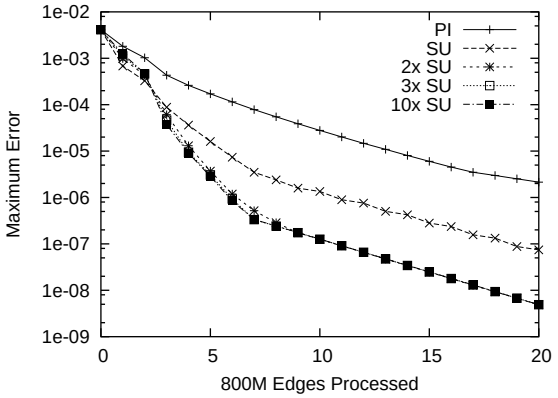

Figure 4: Reiteration: Maximum Error

Additional iterations help maximum error as well, but the diminishing returns are even more pronounced.

Related Work: Arrangement by strongly connected components has appeared several times in various forms. Eiron et al. [4] note that pages with no outlinks form a large fraction of the web and describe how to infer their ranks from the stationary probabilities of a modified graph with these nodes removed. More generally, Arasu et al. [1] and Langville and Meyer [10] view the problem as a block upper triangular linear system, processing strongly connected components in turn. Their techniques focus on decomposing the Markov chain, and require a strict topological order.

These approaches are captured by reiteration over the equivalent grouping. Moreover, UpdateIteration can take advantage of groupings that are only *mostly* topological. This flexibility addresses concerns of the substantial effort needed for data organization and maintenance, and enables grouping by host/domain, which was not possible in the more rigid block triangular techniques.

3.3 Selective Updates

While $z_u = y_u$ is clearly one effective choice, an alternate choice is $z_u = 0$. In effect, we can choose *not* to update node u. Clearly if $y_u = 0$ we need not expend effort to propagate y_u through A, as we will simply be adding zero to several locations in y. Even when y_u is non-zero but small, we may want to defer the update until the gains are more in line with the typical entry. With this in mind, there are various predicates we could use to decide *if* we should process y_u. We will specifically consider:

$$\textbf{Effort}: \text{ Set } z_u = y_u \text{ iff } \frac{|r_u y_u|}{deg_u} \geq \underset{v}{avg}\left\{\frac{|r_v y_v|}{deg_v}\right\},$$

selecting entries with the highest anticipated progress $|r_u y_u|$ per expended effort deg_u. There are other predicates that could be used, each resulting from a different view of which entries are important to process. Examples include choosing those entries with largest *relative* error $|(Px - x)_u|/|x_u|$ or those entries whose range of possible ordinal ranks is largest.

Selective updating has some interesting interaction with sequential and reiterated updates. As we run a pass of sequential updates the average value of $|r_v y_v|/deg_u$ will change, and while we could maintain the average exactly by carefully watching the changes in y, we can also do a more efficient approximation by assuming that $\|y\|_1$ decreases by exactly $r_u|z_u|$. Reiteration is similar, in that each reiteration lowers the weight in a group markedly, by a factor of at least $1-r_u$, but not the average value over all of y. We could base our decisions on the group's average, shrinking with the values we consider, or on the entire average over y.

While the gains of selective updating in terms of computation and memory accesses are clear, savings in terms of disk accesses are less so. It is not possible to skip entries on disk at no cost; data is read from disk in blocks, and the cost is amortized over all entries in the block. Likewise, disk prefetching will prepare subsequent blocks cheaply, and it is unclear that we gain anything by ignoring edge data passed to us. To address this somewhat, it is certainly possible to apply selective updating at a coarser scale than the node level. One could skip entire groups of entries at a time, permitting a volume of edges to be passed over at once and resulting entire disk blocks skipped.

Alternately, Kamvar, Haveliwala, and Golub note in [5] that some pages converge more slowly than others, determining which these are at runtime by observing their relative change in ranks each round. While they use converged values to cull edges associated with converged nodes, we might base a grouping scheme (a la Section 3.2) on convergence rate, determined in a similar manner at run time. Emitting an appropriately grouped edge file can be done efficiently in a single pass so long as the number of groups is not terribly large. This organization of the edge file allows efficient passes over prefixes of the edge file, letting us efficiently process each group at an arbitrary rate. Understanding and experimenting with coarse-grained selective updating and grouping is interesting future research.

Figures 5 and 6 examine the Effort predicate applied to previous schemes (denoted in the figures labels by "E+"). The acceleration we see here is substantial, as selective updating takes advantage of the initially high variability of magnitudes in y. We stress that actual acceleration will be less, although some may be recouped via clever grouping.

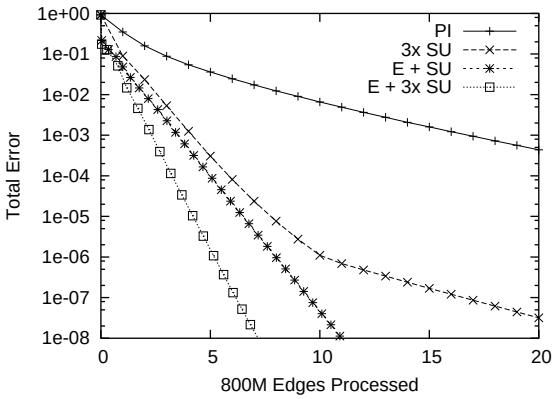

Figure 5: Selective Updating: Total Error

We see substantial acceleration in terms of edges processed, which is, admittedly, a somewhat suspect measure. The message is that the work that *needs* to be done is less.

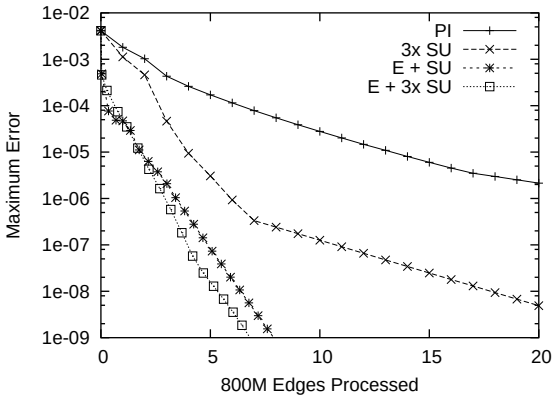

Figure 6: Selective Updating: Maximum Error

Initial acceleration is especially pronounced here, as the selective updates immediately leap two orders of magnitude.

Related Work: Selective updating can be seen in the work of Kamvar, Haveliwala, and Golub [5], who describe a power iteration process wherein entries x_u that appear to have converged are frozen, saving them the recomputation of these entries every iteration. Moreover, edges associated with the frozen nodes are trimmed from the edge file, reducing disk IO required. Freezing is analogous to the non-transmission of an update, and the edge trimming is clearly the basis of the grouping discussed previously, though more final.

The main difference between [5] and UpdateIteration is that in the former the recipient decides whether an update will be propagated, and it is conceivable that significant updates may be ignored. This is particularly evident during incremental changes to the web graph. As the matrix A changes over time, previous stationary distributions prove good starting points for converging to the new stationary distribution. But if only a few links change, entries of x_u not incident to a changed edge will remain stationary after an iteration and will be frozen and never updated.

3.4 Incremental Updates

Over time the adjacency structure of the web changes, and we will want to compute the stationary distribution of a new chain that differs from the old in a relatively small number of locations. The stationary distribution of the old chain is generally viewed as a good first approximation, and indeed it is easy and intelligent to restart PowerIteration on a new chain using the old x.

Restarting UpdateIteration from a specific x appears nontrivial, as we must compute $y = Ax - x + d$, involving a matrix multiplication. In fact, the process is much simpler: Let A and B describe the old and new edge transition matrices, and consider the two associated update vectors y_A and y_B,

$$y_A = Ax - x + d \quad \text{and} \quad y_B = Bx - x + d\,.$$

We can relate the update vector y_B to its antecedent y_A as

$$y_B = y_A + (B - A)x\,.$$

This equivalence shows how to efficiently update y_A to y_B, allowing us to efficiently reinvoke UpdateIteration(B, x, y_B). The effort required in this matrix-vector multiplication is proportional to the number of non-zero entries in $B - A$, corresponding to the number of changed edge weights.

Several approaches to personalization of PageRank are based on personalization of the reset distribution [8, 9], shifting influence to those sites that the distribution favors, and the site linked by them. Recall from Theorem 1 that x converges to the stationary distribution of the chain with reset distribution $d = y - (Ax - x)$. Personalization of the reset distribution is easily performed by incorporating any changes to d into y instead, changing x's limit appropriately. It is worth stressing that x's limit is defined by the distribution *proportional* to d, and we need not worry about renormalizing d if we only make a few changes.

Finally, much of research into Markov chain based ranking research is exploratory: the best setting of weights in A and vectors d and r are not known. Uniform weights seem natural as defaults, but are clearly primitive choices. Exploring link weighting schemes based on content analysis or resetting policies based on content quality require efficient recomputation of ranks. Each of these explorative choices: updating A_{vu}, r_u, and d_u values, are easy in UpdateIteration, corresponding to simple updates to y.

In these three cases above, the changes to the Markov chain often result in sparse updates to y: most of the edges in the graph are stable between recrawls, and much of personalization of reset distributions is localized (upweighting a few trusted/bookmarked pages, for example). In this context, selective updating of Section 3.3 is well suited to efficiently process just those substantial entries, and leave the converged regions of the graph untouched. Of course to accommodate this properly, it makes sense to maintain an edge file of those parts of the graph that experience frequent edge churn, so that we needn't pass over the entire graph.

The fine granularity of sequential updates also allows a very smooth incremental update: we can decompose any update to the adjacency matrix into a set of small updates to the links of each node, which we apply as we visit each node. We need not pause the system to compute $(B - A)x$, but can apply the implications of changes at each node in turn. This becomes all the more relevant in a distributed setting where such pauses could destroy parallelism.

Figure 7 compares various techniques applied to a converged vector y that has had 1000 random positions updated randomly by $\pm 1/n$, emulating either a change in the link structure or reset distribution. For small initial $\|y\|_1$, the scale of the updates does not affect the shape of the curves, so the choice $1/n$ is arbitrary.

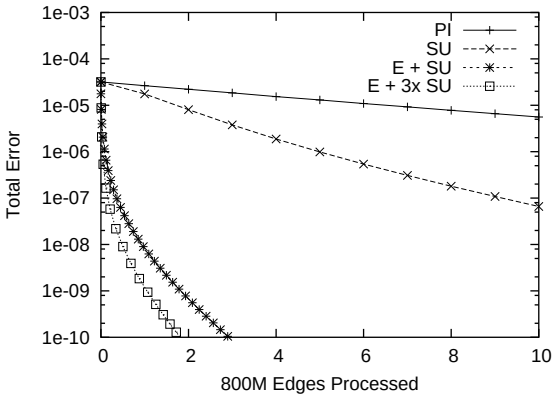

Figure 7: Incremental Updating

It is difficult to characterize the acceleration of the incremental updates by a multiplicative factor, as it is clearly a different shape than the standard curves. Several orders of magnitude are gained immediately, with the slope arriving at the shape of Figure 5 as the initially concentrated y vector is distributed more uniformly.

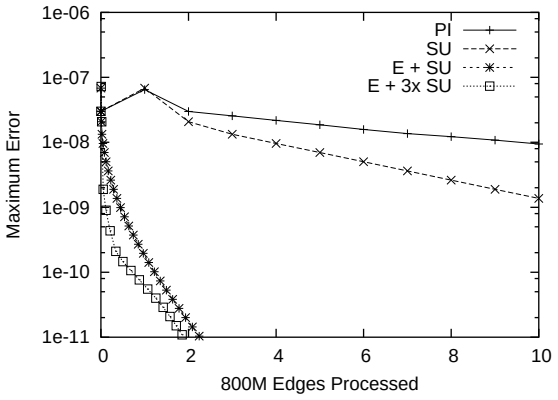

Figure 8: Incremental Updating

Maximum error exhibits the same behavior as total error, dropping rapidly as the initially sparse vector is dispersed. The initial hiccup again reflects the sensitive nature of the maximum error measure.

Related Work: Chien et al. [3] describe an approach to incremental updating that is based on the construction and analysis of a new a Markov chain on nodes within a modest neighborhood of the graph changes, and a supernode representing nodes outside this horizon. Their approach is similar in spirit to ours, in that attention is restricted to the relatively small region where change may occur. However, rather than fix a region and degree of accuracy, UpdateIteration discovers where updates are needed as it goes, accommodating any degree of accuracy fluidly.

Haveliwala [8] and Jeh and Widom [9] have done work on efficient personalization, observing that the function mapping reset distributions to stationary distributions is linear. This enables very efficient manners of synthesizing personalized PageRanks from a set of precomputed PageRanks based on various reset distributions. For example, a page's d_u value can be increased by folding in the stationary distribution of a random walk that resets to only that page, exactly analogous to increasing and propagating y_u.

3.5 Floating Point Implications

Our ability to choose z arbitrarily has implications for floating point error. We have the flexibility to always choose z_u to be a power of two, so that its addition to x will result in nominal floating point loss. This is harder to guarantee with y, as transmission along weighted edges will change z_u from a power of two. Understanding and improving floating point behavior has positive implications for the introduction of strength reduction and low precision arithmetic, of particular interest in this setting where maintaining all of x or y in memory is challenging. Additionally, UpdateIteration propagates and combines updates z_u, which are typically of smaller magnitude than the x_u values that PowerIteration operates with, and the precision maintained is thus higher.

3.6 Distribution and Robustness

If we remove the sequential behavior from sequential updates, we see that updates in UpdateIteration can actually be totally asynchronous. Moreover, our choices for z_u are made locally with only a modicum of global information. This allows for a very smooth distributed implementation, in which the only coordination between compute nodes that is required is eventual communication of the updates applied. We can delay and reorder inter-node z transmissions until the updates are significant, batching and trimming network overhead. Clearly, the best update schedule is highly dependent on the system topology, and we refrain from giving explicit suggestions here.

In an extreme case of delay, a compute node may be unavailable for a long period of time or even crash. Other compute nodes can continue in its absence, functioning under the belief that the PageRanks associated with that compute node simply have not changed. If the node comes online again it simply reenters the computation, transmitting and receiving updates. As noted for incremental updates, the granularity of sequential updating is very fine, and the amount of work needed to roll forward from any checkpoint can be arbitrarily small.

3.7 Decentralization

The Markov chain we have studied simulates the propagation of probability mass through a directed communication network whose nodes happen to be computational agents. The propagation of updates is easily performed within the communication network, as updates are only transmitted along links. The initial values of $x = 0$ and $y = d$ are easily chosen, as d needn't be normalized. As the network changes, in the incremental fashion suggested by Section 3.4, the necessary updates to y are computable by the source of the edge that has arrived or departed. Gracefully departing nodes removing their incoming edges using this update mechanism and apply the update $z_u = y_u - d_u$ before departing.

4. CONCLUSIONS AND FUTURE WORK

We have examined an algorithmic reformulation of the traditional power iteration algorithm based on the propagation of updates rather than values. UpdateIteration enables several algorithmic optimizations that result in more efficient convergence. Moreover, the optimizations are well suited to the problems of incremental and personalized updates to the underlying Markov chain, and permit flexible operation in a distributed setting.

The optimizations presented here are likely just a sampling of what can be done to accelerate computation of PageRank. These optimizations are intended to take advantage of particular features of computer systems, and it seems likely that other features may yet be exploited, both for performance and potentially quality of ranking. Techniques such as Arnoldi Iteration and unsymmetric Lanczos are tempting targets, as is the power extrapolation approach of Kamvar et al. [7]. Additionally, there is work to be node exploring the new possibilities enabled through efficient PageRank computation.

5. ACKNOWLEDGMENTS

The author would like to thank several people who contributed constructive ideas and observations. Michael Isard, Steve Chien, Kevin McCurley, the participants of the Workshop on Search and Meta-Search, and the anonymous reviewers all gave valuable comments which have greatly improved the presentation.

6. REFERENCES

[1] Arvind Arasu, Jasmine Novak, Andrew Tomkins, and John Tomlin, Pagerank computation and the structure of the web: Experiments and algorithms. *WWW 2002* poster.

[2] Sergey Brin and Lawrence Page, The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks* 30(1-7): 107-117 (1998).

[3] Steve Chien, Cynthia Dwork, Ravi Kumar, Dan Simon, and D. Sivakumar, Link evolution: Analysis and algorithms. *Internet Mathematics*: Volume 1, No. 3, pp. 277-304.

[4] Nadav Eiron, Kevin McCurley, and John Tomlin, Ranking the web frontier. *WWW 2004*.

[5] Sepandar Kamvar, Taher Haveliwala, and Gene Golub, Adaptive Methods for the Computation of PageRank. Stanford University Technical Report, 2003.

[6] Sepandar Kamvar, Taher Haveliwala, Christopher Manning, and Gene Golub, Exploiting the Block Structure of the Web for Computing PageRank. Stanford University Technical Report, 2003.

[7] Sepandar Kamvar, Taher Haveliwala, Christopher Manning, and Gene Golub, Extrapolation Methods for Accelerating PageRank Computations. *WWW 2003*.

[8] Taher Haveliwala, Topic-Sensitive PageRank. *WWW 2002*.

[9] Glen Jeh and Jennifer Widom, Scaling Personalized Web Search. *WWW 2003*.

[10] Amy Langville and Carl Meyer, A Reordering for the PageRank problem. NCSU CRSC Technical Report #CRSC-TR04-16. March 2004.

7. APPENDIX A: PROOFS

We now look at the two deferred proofs from Section 2. Recall that Theorem 1 requires the entries of d be non-negative.

PROOF OF THEOREM 1. $Px-x$ and $y = Ax-x+d$ differ only in the amount of d added to $Ax-x$. We can thus write their difference as

$$y - (Px - x) \quad = \quad d - dr^T x/\|d\|_1 \ . \tag{1}$$

Summing the coordinates of vectors on both sides of (1), and noting that $\sum_u (Px)_u = \sum_u x_u$ and $\sum_u d_u = \|d\|_1$, gives

$$\sum_u y_u \quad = \quad \|d\|_1 - r^T x \ . \tag{2}$$

To prove the first stated inequality, we move y to the right hand side of (1), take norms, and use the triangle inequality.

$$\|Px - x\|_1 \quad \leq \quad \|y\|_1 + |(\|d\|_1 - r^T x)| \ . \tag{3}$$

Substituting $\sum_u y_u$ for $\|d\|_1 - r^T x$ and then $|\sum_u y_u| \leq \|y\|_1$,

$$\|Px - x\|_1 \quad \leq \quad \|y\|_1 + |\sum_u y_u| \quad \leq \quad 2\|y\|_1 \ . \tag{4}$$

Similarly, the second stated inequality results from the inequalities

$$\|x\|_1 \ \geq \ r^T x \ = \ \|d\|_1 - \sum_u y_u \ \geq \ \|d\|_1 - \|y\|_1 \ . \tag{5}$$

with the inequality $\|x\|_1 \geq r^T x$ following as all $|r_u| \leq 1$. □

The proof of Theorem 2 relies on the assumption that the coordinates of z lie between zero and the corresponding y_u.

PROOF OF THEOREM 2. As each z_u lies between zero and y_u, we have that $\|y - z\|_1 = \|y\|_1 - \|z\|_1$, and starting from the triangle inequality

$$\|y + Az - z\|_1 \quad \leq \quad \|y - z\|_1 + \|Az\|_1 \tag{6}$$

$$\quad = \quad \|y\|_1 - \|z\|_1 + \|Az\|_1 \tag{7}$$

Column u of A sums to r_u, and thus $\|Az\|_1 \leq \sum_u (1-r_u)|z_u|$.

$$\|y + Az - z\|_1 \quad \leq \quad \|y\|_1 - \sum_u |z_u| + \sum_u (1 - r_u)|z_u| \tag{8}$$

Collecting the summands yields the claimed bound. □