Efficient Computation of PageRank

Taher H. Haveliwala[*]
Stanford University
taherh@db.stanford.edu

October 18, 1999

Abstract

This paper discusses efficient techniques for computing PageRank, a ranking metric for hypertext documents. We show that PageRank can be computed for very large subgraphs of the web (up to hundreds of millions of nodes) on machines with limited main memory. Running-time measurements on various memory configurations are presented for PageRank computation over the 24-million-page Stanford WebBase archive. We discuss several methods for analyzing the convergence of PageRank based on the induced ordering of the pages. We present convergence results helpful for determining the number of iterations necessary to achieve a useful PageRank assignment, both in the absence and presence of search queries.

1 Introduction

The dramatic growth of the world-wide web, now exceeding 800 million pages [11], is forcing modern web search engines to look beyond simply the content of pages in providing relevant answers to queries. Recent work in utilizing the link structure of the web for improving the quality of search results is promising. In particular, the Google search engine uses PageRank, an iterative algorithm that determines the "importance" of a web page based on the "importance" of its parent pages [5]. A related algorithm used by the IBM HITS system maintains a *hub* and an *authority* score for each page, in which the authority score for a page is determined by the hub scores of its parents, and the hub score of a page is determined by the authority scores of its children [10].

Given that centralized search engines are having trouble simply indexing the entire web [7, 11], presenting users with search-query results relevant to them is becoming increasingly difficult. The growth of the web will force a greater reliance on client-side filtering and relevance analysis, and *personalized* searching techniques become essential. Part of our research in personalizing link-based ranking algorithms involves applying biases during the iterations to increase the importance of certain categories of pages [4]. For instance, we can influence PageRank to weight pages related to computers more heavily than pages related to cars. In this scenario, each user would

[*]Supported by NSF IIS-98-11904 and NSF Graduate Research Fellowship

be expected to run PageRank biased with their personalization profile on their own personal machines, with the necessary link structure provided to them on inexpensive media such as DVD-ROM. Reducing the running times and memory requirements of personalized ranking algorithms is essential.

After reviewing the PageRank algorithm [4, 5] in Section 2, we discuss the various aspects of its running time, and show that it can be computed efficiently:

- We describe an implementation, using the principle underlying the block nested-loops-join strategy, that efficiently controls the cost per iteration even in low memory environments (Section 3)
- We empirically show that single-precision rank values are sufficient for the computation (Section 4)
- We investigate techniques for determining the number of iterations required to yield a useful PageRank assignment. We present convergence results using several metrics that are useful when the induced ordering on pages, rather than the PageRank value itself, is considered. We investigate the convergence of the globally induced ordering, as well as the ordering induced over specific search results (Section 5)

By showing that a useful ranking can be computed for large web subgraphs in as little as an hour, with only modest main-memory requirements, the potential for personalization is made clear.

2 Review of PageRank

First let us review the motivation behind PageRank [5]. The essential idea is that if page u has a link to page v, then the author of u is implicitly conferring some importance to page v. How much importance does u confer? Let N_u be the outdegree of page u, and let $Rank(p)$ represent the importance (i.e., PageRank) of page p. Then the link (u, v) confers $Rank(u)/N_u$ units of rank to v. This simple idea leads to the following fixpoint computation that yields the rank vector $Rank^*$ over all of the pages on the web. If N is the number of pages, assign all pages the initial value $1/N$. Let B_v represent the set of pages pointing to v. In each iteration, propagate the ranks as follows:[1]

$$\forall_v Rank_{i+1}(v) = \sum_{u \in B_v} Rank_i(u)/N_u$$

We continue the iterations until $Rank$ stabilizes to within some threshold. The final vector $Rank^*$ contains the PageRank vector over the web. This vector is computed only once after each crawl of the web; the values can then be used to influence the ranking of search results [2].

The process can also be expressed as the following eigenvector calculation, providing useful insight into PageRank. Let M be the square, stochastic matrix corresponding to the directed graph G of the web, assuming all nodes in G have at least

[1]Note that for $u \in B_v$, the edge (u, v) guarantees $N_u \geq 1$

one outgoing edge. If there is a link from page j to page i, then let the matrix entry m_{ij} have the value $1/N_j$. Let all other entries have the value 0. One iteration of the previous fixpoint computation corresponds to the matrix-vector multiplication $M \times Rank$. Repeatedly multiplying $Rank$ by M yields the dominant eigenvector $Rank^*$ of the matrix M. Because M corresponds to the stochastic transition matrix over the graph G, PageRank can be viewed as the stationary probability distribution over pages induced by a random walk on the web.

We can measure the convergence of the iterations using the *Residual* vector. Because M is stochastic (i.e., the entries in each column sum to 1), the dominant eigenvalue of M is 1. Thus the PageRank vector $Rank^*$, the dominant eigenvector of M, has eigenvalue 1, leading to the equality $M \times Rank^* = Rank^*$. We can use the deviation from this equality for some other vector as an error estimate. For some intermediate vector $Rank_i$, let $Residual_i = M \times Rank_i - Rank_i$. Equivalently, because multiplication by M is an iteration step, we have $Residual_i = Rank_{i+1} - Rank_i$. We can treat $||Residual_i||$ as an indicator for how well $Rank_i$ approximates $Rank^*$. We expect $||Residual_i||$ to tend to zero after an adequate number of iterations.

We now address several issues regarding the computation. We previously made the assumption that each node in G has at least one outgoing edge. To enforce this assumption, we can iteratively remove nodes in G that have outdegree 0. Alternatively, we can conceptually add a complete set of outgoing edges to any node with outdegree 0. Another caveat is that the convergence of PageRank is guaranteed only if M is irreducible (i.e., G is strongly connected) and aperiodic [12]. The latter is guaranteed in practice for the web, while the former is true if we add a dampening factor to the rank propagation. We can define a new matrix M' in which we add transition edges of probability $\frac{1-c}{N}$ between every pair of nodes in G:

$$M' = cM + (1-c) \times [\frac{1}{N}]_{N \times N}$$

This modification improves the quality of PageRank by introducing a decay factor which limits the effect of rank *sinks* [4], in addition to guaranteeing convergence to a unique rank vector. For the above modification to M, an iteration of PageRank can be expressed as follows:[2]

$$M' \times Rank = cM \times Rank + (1-c) \times [\frac{1}{N}]_{N \times 1}$$

We can bias PageRank to weight certain categories of pages by replacing the uniform vector $[\frac{1}{N}]_{N \times 1}$ with the nonuniform $N \times 1$ personalization vector $\vec{p}$ as discussed in [4]. In terms of the random-walk model of PageRank, the personalization vector represents the addition of a complete set of transition edges where the probability of edge (u, v) is given by $(1-c) \times p_v$. Although the matrix M' that results from the modifications discussed above is not sparse, we never need to store it explicitly. We need only the ability to evaluate $M' \times Rank$ efficiently.

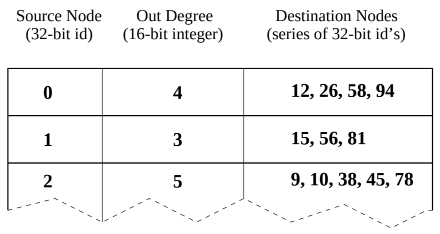
[2]This equality makes use of the fact that $||Rank||_1 = 1$

3 Efficient Memory Usage

If we momentarily ignore the scale of the web, the implementation of PageRank is very simple. The sheer scale of the web, however, requires much greater care in the use of data structures. We will begin with a detailed discussion of a naive implementation, during which we will clarify the size of the datasets. The naive algorithm is useful for gaining a clear understanding of matrix-vector multiplication in the specific context of PageRank. We will then present an efficient version of the algorithm, reminiscent of the block nested-loops-join algorithm, that substantially lowers the main memory requirements. Similar strategies for fast in-memory matrix-vector multiplies are commonly used in the scientific computing community for improving caching behavior [9, 14]. As we are dealing with massive web repositories in which the data structures are measured in gigabytes and are stored on disk, we will take a more i/o-centric approach in presenting the algorithm and its cost analysis. Empirical timing results for both the naive and block-based approaches are presented.

3.1 The Naive Technique

The Stanford WebBase, our local repository of the web, currently contains roughly 25 million pages. There are over 81 million urls in the link graph, including urls that were not themselves crawled, but exist in the bodies of crawled pages. For our experiments, we first used a preprocessing step that removed *dangling* pages, meaning pages with no children. Starting with the 81-million-node graph, all nodes with outdegree 0 were removed. The step was repeated once more on the resulting graph, yielding a subgraph with close to 19 million nodes. This process was needed since the original graph is a truncated snapshot of the web with many dangling nodes. Nodes with no outgoing links pointing back to the crawled subgraph can adversely affect the PageRank assignment, as mentioned in Section 2. The node id's were assigned consecutively from 0 to 18,922,290. The link structure for the final graph, referred to as *Links*, is stored on disk in a binary format, illustrated textually in Figure 1. The source-id and each of the destination-id's are stored as 32-bit integers.



Source Node (32-bit id)	Out Degree (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	**4**	**12, 26, 58, 94**
1	**3**	**15, 56, 81**
2	**5**	**9, 10, 38, 45, 78**

Figure 1: The Link Structure

The outdegree is stored as a 16-bit integer. The size of the link structure, after the preprocessing steps mentioned above, is 1.01 GB, and is assumed to exceed the size of main memory.

The setup for the naive PageRank implementation is as follows. We create two arrays of floating point values representing the rank vectors, called *Source* and *Dest*, as conceptually shown in the matrix-vector multiplication given in Figure 2. Each

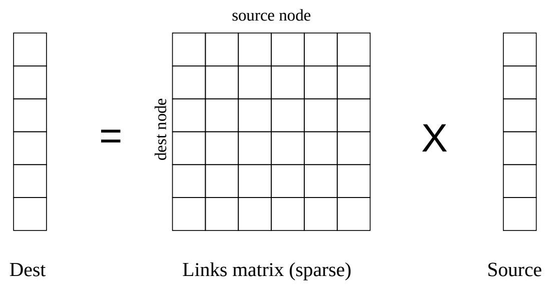


Figure 2: The Rank Vectors

vector has N entries, where N is the number of nodes in our web graph. In our experiment, N was 18,922,290. The rank values for iteration i are held in $Source$, and the rank values for iteration $i+1$ are constructed in $Dest$. Using single-precision values, these arrays for our particular graph have a combined size of over 150 MB. The iteration steps can be expressed as follows (where c is the dampening factor):

```
$\forall_s Source[s] = 1/N$
while($residual > \tau$) {
    $\forall_d Dest[d] = 0$
    while(not $Links$.eof()) {
        $Links$.read($source$, $n$, $dest_1, dest_2, ..., dest_n$)
        for $j = 1 \dots n$
            $Dest[dest_j] = Dest[dest_j] + Source[source]/n$
    }
    $\forall_d Dest[d] = c \times Dest[d] + \frac{1-c}{N}$ /* dampening or personalization*/
    $residual = ||Source - Dest||$ /* recompute only every few iterations */
    $Source = Dest$
}
```

We make successive passes over $Links$, using the current rank values, held in $Source$, to compute the next rank values, held in $Dest$. We can stop when the norm of the difference between $Source$ and $Dest$ reaches some threshold, or alternatively after a prechosen number of iterations. Assuming main memory is large enough to hold $Source$ and $Dest$, the i/o cost for each iteration of the above implementation is given by:

$$C = |Links|$$

If main memory is large enough to hold only the $Dest$ array, and we assume that the link structure is sorted on the source field, the i/o cost is given by:

$$C = |Source| + |Dest| + |Links|$$

$Source$ needs to be sequentially read from disk during the rank propagation step, and $Dest$ needs to be written to disk to serve as the $Source$ vector for the subsequent iteration.

Although many machines may have enough main memory to hold these arrays, a larger crawl with 50 to 100 million pages clearly will result in rank vectors that exceed the main memory of most computers. Considering that the publicly indexable web now contains roughly 800 million pages [11], the naive approach is infeasible for large subgraphs of the web. As mentioned above, if the link structure is sorted on the source field, the accesses on *Source* will be sequential, and will not pose a problem. However, the random access pattern on the *Dest* array leads the working set of this implementation to equal the size of the *Dest* array. If the main memory cannot accommodate the *Dest* array, the running time will increase dramatically and the above cost analysis becomes invalid.

3.2 The Block-Based Strategy

There is a similarity between an iteration of PageRank and the relational *join* operator. Let the children of node *source* in the structure *Links* be represented by $CHILDREN(source, Links)$. If we consider the two rank arrays *Source* and *Dest* as relations, an iteration of PageRank is performing a join in which each entry *source* of *Source* joins with each entry *dest* of *Dest* if $dest \in CHILDREN(source, Links)$. Instead of adjoining fields of two tuples, however, we are adding in the (scaled) value of the *Source* entry to the *Dest* entry.

Although the analogy is not exact, the core technique used by the block-oriented join strategy can be used to control the working set of the PageRank algorithm as well. We will partition the *Dest* array, the cause of the large working set, into β blocks each of size D pages, as illustrated in Figure 3. If P represents the size of main memory

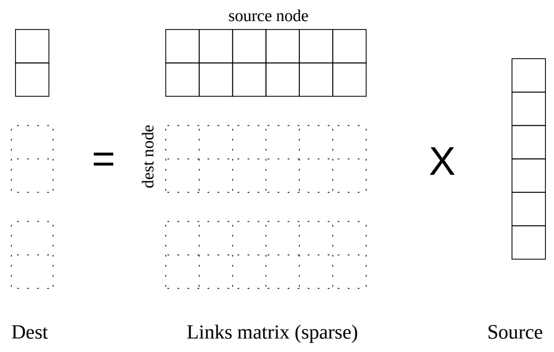
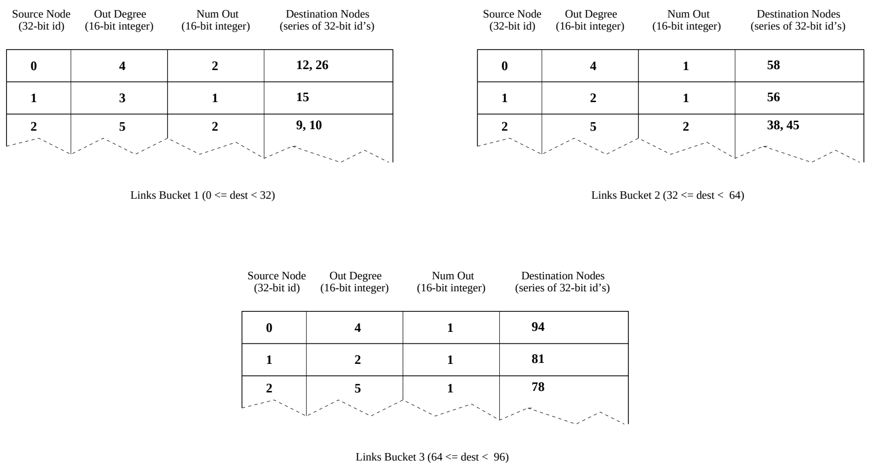


Figure 3: Blocked Multiplication

in physical memory pages, then we require $D \leq P - 2$, since we must leave input buffers for reading in *Source* and *Links*. The links file *Links* must be rearranged to reflect this setup. We partition *Links* into β links files $Links_0, \ldots, Links_{\beta-1}$, such that the *destinations* field in $Links_i$ contains only those nodes *dest* such that $\beta \times i \leq dest < \beta \times (i+1)$. In other words, the outgoing links of a node are bucketed according to the range that the identifier of the destination page falls into. The partitioning of the links of Figure 1 for the case of three blocks is shown in Figure 4.

This partitioning scheme obeys the equality:

$$CHILDREN(source, Links) = \bigcup_i CHILDREN(source, Links_i)$$



Source Node (32-bit id)	Out Degree (16-bit integer)	Num Out (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	4	2	12, 26
1	3	1	15
2	5	2	9, 10

Links Bucket 1 (0 <= dest < 32)

Source Node (32-bit id)	Out Degree (16-bit integer)	Num Out (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	4	1	58
1	2	1	56
2	5	2	38, 45

Links Bucket 2 (32 <= dest < 64)

Source Node (32-bit id)	Out Degree (16-bit integer)	Num Out (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	4	1	94
1	2	1	81
2	5	1	78

Links Bucket 3 (64 <= dest < 96)

Figure 4: Partitioned Link File

Note that $\sum |Links_i| > |Links|$ because of the extra overhead caused by the redundant storage of the source node and outdegree entries in each of the partitions. The block-oriented algorithm proceeds as follows:

```
$\forall_s Source[s] = 1/N$
while($residual > \tau$) {
    for $i = 0 \ldots \beta - 1$ {
        $\forall_d Dest_i[d] = 0$
        while(not $Links_i$.eof()) {
            $Links_i$.read($source$, $n$, $k$, $dest_1, dest_2, ..., dest_k$)
            for j = $1 \ldots k$
                $Dest_i[dest_j] = Dest_i[dest_j] + Source[source]/n$
        }
        $\forall_d Dest_i[d] = c \times Dest_i[d] + \frac{1-c}{N}$ /* dampening or personalization*/
        Write $Dest_i$ to disk
    }
    $residual = ||Source - Dest||$ /* recompute only every few iterations */
    $Source = Dest$
}
```

Because $Links_i$ is sorted on the *source* field, each pass through $Links_i$ requires only one sequential pass through $Source$. The working set of this algorithm is exactly P by design, so no swapping occurs. Define ϵ to be such that the equality $\sum_i |Links_i| = |Links| \times (1+\epsilon)$ is satisfied. The cost of this approach is then given by:

$$C = \beta \times |Source| + |Dest| + |Links| \times (1+\epsilon)$$

In practice, ϵ is reasonably small, as shown in Figure 7. The other cost introduced by the partitioning scheme is the need to make β passes over the $Source$ array. However, since in practice we have $|Links| > 5 \times |Source|$, this additional overhead is reasonable. Note that computing the residual during every iteration would require an additional pass over *Source*, which is not included in the above cost analysis. We can largely avoid this cost by computing the residual only at fixed intervals.

If we had stored the links in transpose format, in which each entry contained a node and a list of parent nodes, then the above algorithm would remain essentially the same, except that we would break the *Source* array into β blocks, and make multiple passes over the *Dest* array. We would successively load in blocks of *Source*, and fully distribute its rank according to $Links_i^T$ to all destinations in *Dest*. However, note that each "pass" over the *Dest* array requires reading in the values from disk, adding in the current *Source* block's contributions, and then writing out the updated values to disk. Thus using $Links^T$ rather than *Links* incurs an additional i/o cost of $|Dest|$, since *Dest* is both read and written on each pass.

In order to take advantage of sequential read transfer benefits, we can load in more than one page of *Source* and $Links_i$ at a time as we stream them in. This buffering reduces the effective time required per pass of *Source* and *Links*, at the expense of increasing β. The best strategy for allocating memory between the three data structures is dependent on the machine and disk architecture, although any reasonable allocation will allow the PageRank algorithm to be used efficiently in cases where the *Rank* vectors exceed main memory.

3.3 Timing Results

For our experiments, we used a 450MHz Pentium-III machine with a 7200-RPM Western Digital Caviar AC418000 hard disk. We measured the running times of PageRank over roughly 19 million pages using three different partitionings of the *Dest* array: 1-block (i.e., naive), 2-blocks, and 4-blocks. The expected memory usage is given in Figure 5. We tested the three partitioning strategies on three different memory configurations: 256 MB, 64 MB, and 32 MB.

The time required per iteration of PageRank is given for each of the three partitionings under each of the three memory configurations in Figure 6. As expected, the most efficient strategy is to partition the *Dest* array (and the corresponding *Links* structure) into enough blocks so that a single *Dest* block can fit in physical memory. Using too many blocks slightly degrades performance, as both the number of passes over *Source* and the size of *Links* increase. Figure 7 shows the total size of the link structure for the three partitionings, as well as the associated ϵ, as discussed in Section 3.2. Using too few blocks, however, degrades performance by several orders of magnitude. For the cases in Figure 6 where the block size exceeds physical memory, we had to estimate the full iteration time from a partially completed iteration, as the running times were unreasonably high.

The blocking strategy commonly used for other algorithms, including the relational join operator, is very effective in controlling the memory requirements of PageRank. The block-based PageRank is not an approximation of normal PageRank: the same matrix-vector multiplication $M' \times Source$ is performed whether or not *Dest* and *Links* are partitioned. The resultant PageRank vector is identical regardless of the number of blocks used. As the Stanford WebBase increases in size to several hundreds of

Number of Blocks	Expected Working Set
1	72 MB
2	36 MB
4	18 MB

Figure 5: Expected Memory Usage

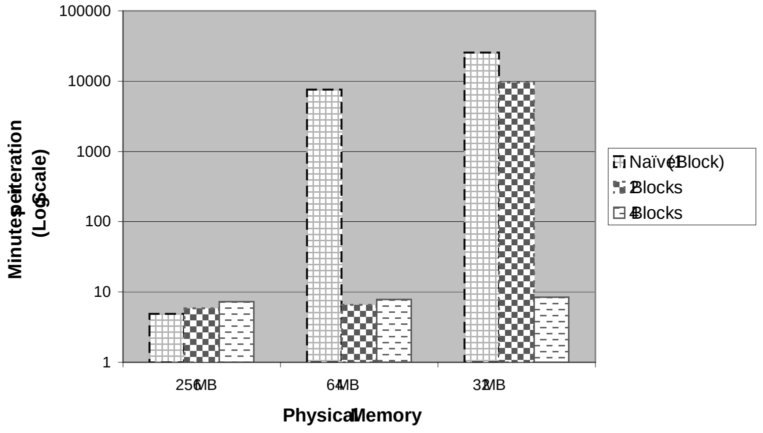


Figure 6: Log Plot of Running Times

Number of Blocks	Size of Links	∈
1	1.01 GB	0
2	1.14 GB	0.13
4	1.29 GB	0.28

Figure 7: Link Structure Growth

millions of pages, the block-oriented technique will be essential in computing PageRank, even on machines with fairly large amounts of main memory. Furthermore, the block-oriented technique will be necessary for allowing individual users to compute their own personalized PageRank.

4 Accuracy

When computing PageRank, we can use either single-precision or double-precision values for the *Source* and *Dest* arrays. Using double-precision for *Source* and *Dest*, however, would adversely affect the running time by doubling the sizes of the two vectors. Here we show that single-precision *Source* and *Dest* vectors are sufficient. Double-precision values should be used for any individual variables, such as the current residual or the current total rank. These individual variables of course have negligible impact on the memory footprint.

The use of single-precision *Rank* vectors does not lead to significant numerical error. We computed PageRank for 100 iterations using single-precision *Source* and *Dest* vectors. We converted the final computed PageRank vector to the equivalent double-precision vector, and performed a double-precision iteration step to get an accurate value for the residual: 2.575×10^{-4}. We then recomputed PageRank for 100 iterations using exclusively double-precision vectors, but found that the residual of the final vector had not noticeably improved: 2.571×10^{-4}.

5 Convergence Analysis

Although PageRank is guaranteed to converge given the conditions mentioned in Section 2, there are several measures we can use to analyze the convergence rate. The residual, which is the norm of the difference of the PageRank vectors from two successive iterations as discussed in Section 2, is one possible measure of the convergence. A more useful approach to analyzing the convergence involves looking at the *ordering* of pages induced by the *Rank* vector. If the PageRank values will be used strictly for determining the *relative* importance of pages, the convergence should be measured based on how the ordering changes as the number of iterations increases. In this section, we will discuss various techniques for analyzing the convergence of induced orderings. Depending on the final application of the rankings, we can concentrate on the ordering induced over all pages or on the ordering induced over results to specific queries. We will first discuss global orderings, and then look at query specific orderings.

5.1 Global Ordering

The global ordering induced by PageRank provides an insightful view into the convergence rate. We analyze the convergence of the induced ordering in two ways: a histogram of the difference in positions of pages within two orderings, and a similarity measure between two ordered listings.

When analyzing the change in page ordering while varying the number of iterations, we are more concerned with instability among the top ranked pages; whether

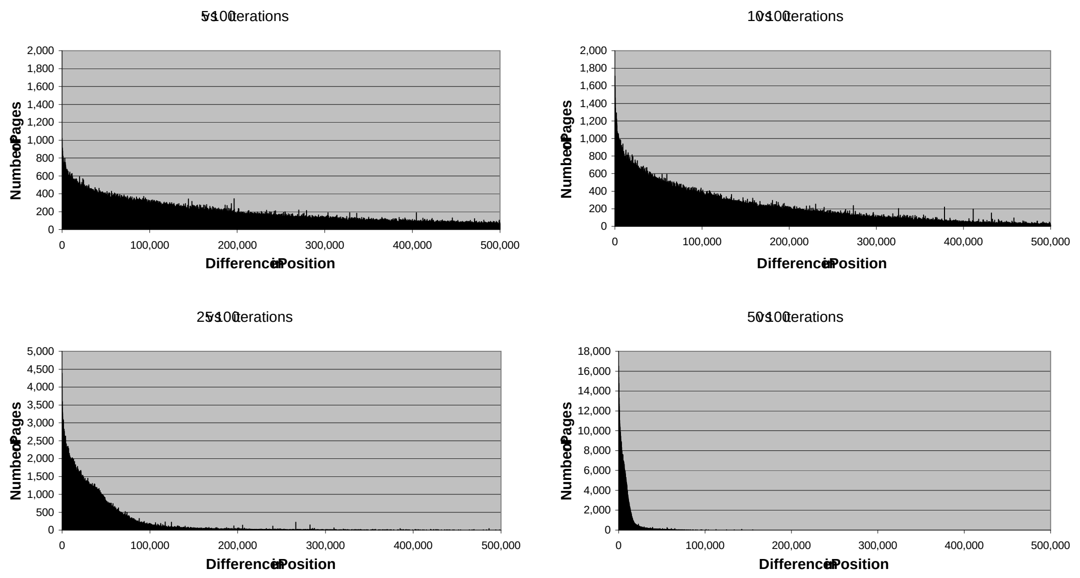


Figure 8: Histograms of Position Difference

or not we deem that a particular page is the least important or the second least important is usually irrelevant.

Figure 8 shows histograms, with a bucket size of 100, of position differences for the orderings induced by various numbers of iterations, when compared with the ordering induced by 100 iterations. We considered a page if and only if at least one of the orderings placed it among the top 1 million, to account for our intuition that the ordering among highly ranked pages is more significant. For the ordering induced by 50 iterations, we see that the bulk of pages occur at similar positions as in the ordering induced by 100 iterations.

We now turn to the similarity measure. In many scenarios, we are only concerned with identifying the top pages – we may not necessarily need an exact ordering among them. For instance, if we only have the resources to index (or otherwise process) a small subset of the web, we might choose to concentrate on the (unordered) set of the most highly ranked pages. We define the *similarity* of two sets of pages A and B to be $\frac{|A \cap B|}{|A \cup B|}$. To visualize how closely two ordered rankings agree on identifying the top pages, we successively compute the similarity among the top n pages according to each ordering, as n is increased in stepped increments. Figure 9 is a graph of the similarity of orderings as n is increased, in steps of 100, to 1 million pages. Here we see that the ordering induced by only 25 iterations agrees very closely with the ordering induced by 100 iterations on what the top pages are.

5.2 Query Specific Ordering

PageRank is currently used predominantly for the ranking of search results. Although analyzing the global ordering is useful in certain applications, we show here that much of the instability in the ordering across iterations tends to affect the relative rankings

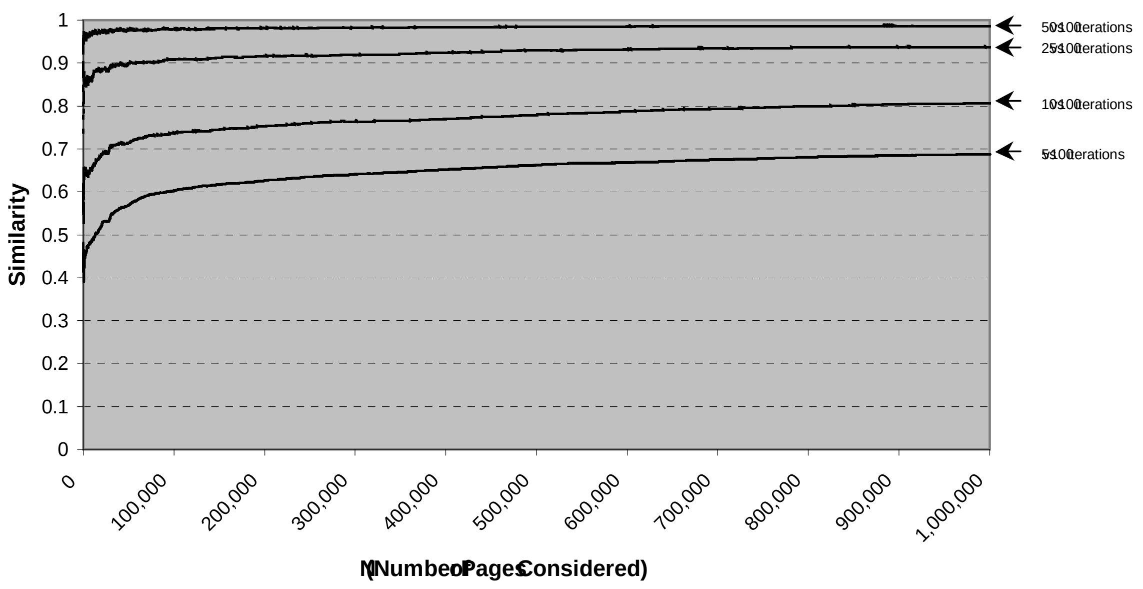


Figure 9: Global Similarity

only of unrelated pages. Just as we are often more concerned with instability among highly ranked pages, we are also often more concerned with instability among pages that are likely to co-occur in the results of a search query.

As before, PageRank is computed only once, yielding a global ordering over web pages. When investigating query-specific ordering, however, we take a subset of these pages, corresponding to the results of a conjunctive search query, and analyze the relative ordering of pages within this result set. Here we analyze the convergence for the induced orderings of the results of two typical search queries. For the *games* query, we retrieved the urls of all pages in the WebBase that contained the words 'action', 'adventure', 'game', 'role', and 'playing'. For the *music* query, we retrieved the urls of all pages that contained the words 'music' and 'literature'. Out of the roughly 20 million pages used for the query, 1,567 urls satisfied the *games* query, and 19,341 urls satisfied the *music* query. We can see in Figures 10 and 11 that the orderings induced by only 10 iterations agree fairly well with the orderings induced by 100 iterations. The PageRank process seems to converge very quickly when viewed in the context of query results.

The residual vector $M' \times Rank_i - Rank_i$, while appealing in a mathematical sense, is only a myopic view of the convergence rate. Looking at the induced ordering provides a more practical view of the convergence behavior. The techniques we discussed are applicable in a broader context as well – we have used them to measure the difference in orderings induced by various personalizations of PageRank, in which the number of iterations remains constant.

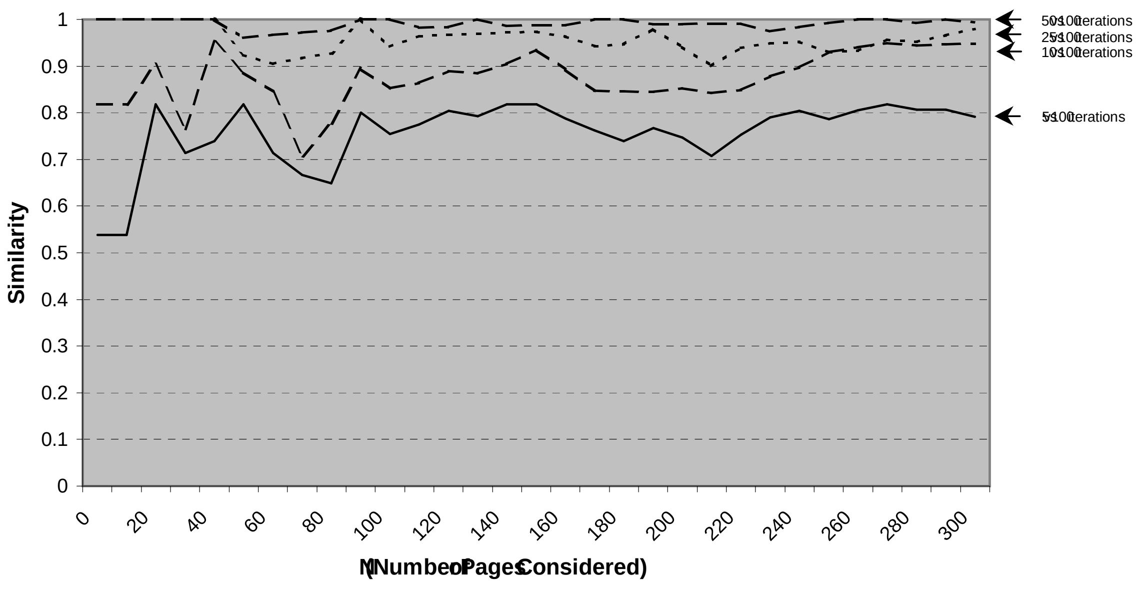


Figure 10: Games Query Similarity

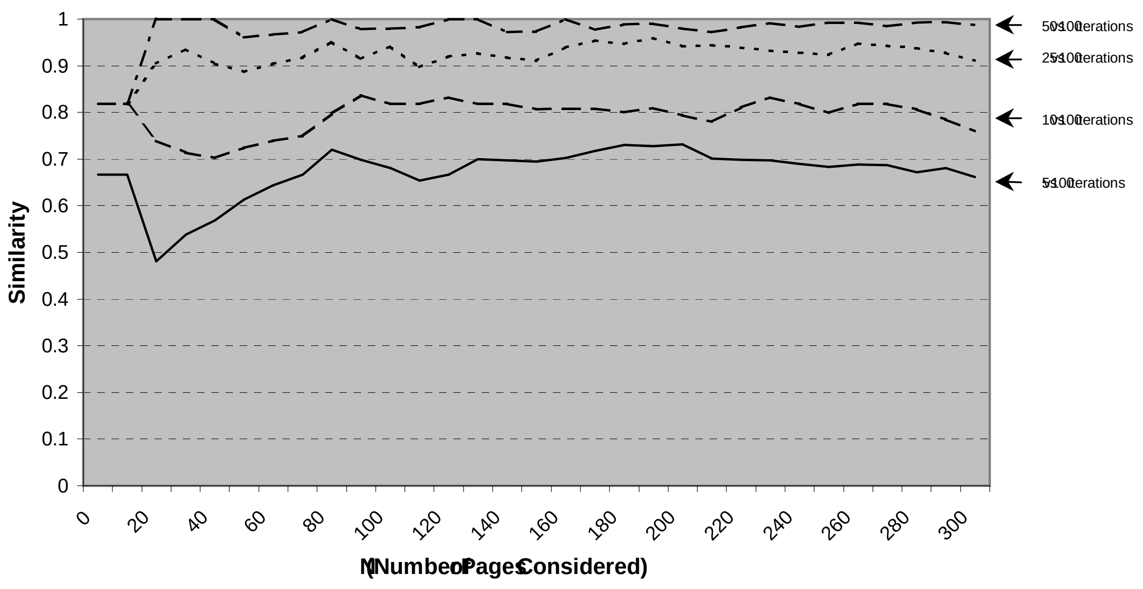


Figure 11: Music Query Similarity

6 Future Work

Personalization of web-based information retrieval will become an increasingly important area of research as the amount of available data on the web continues to grow. We have shown that PageRank can be computed efficiently on modestly equipped machines, suggesting that individual users can compute their own *personalized* PageRank vectors. We will be investigating the most effective means of constructing the personalization vector, as discussed in Section 2, so that the resultant ranking vector best captures the *user's* notion of the importance of a page. We envision utilizing a user's browsing history and bookmark collection to build the personalization vector.

In Section 5.2 we empirically showed that an accurate PageRank vector can be computed in as few as 10 iterations, if accuracy is measured in the context of the induced ordering of results to conjunctive search queries. This convergence result suggests further experiments to determine to what extent the exact nature of the query affects how many iterations are needed before the induced ordering of the query result stabilizes. Using the methods discussed in this paper for analyzing the effect on induced ordering, we will further explore techniques for speeding up PageRank computation while minimizing the loss of accuracy.

7 Conclusion

Algorithms harnessing the link structure of the web are becoming increasingly useful as tools to present relevant results to search queries. Although the PageRank algorithm is based on a simple idea, scaling its implementation to operate on large subgraphs of the web requires careful arrangement of the data. We have demonstrated that PageRank can be run on modestly equipped machines. We have determined that single-precision *Rank* vectors are sufficient for accurate computation. We have presented several ways of analyzing the convergence of the algorithm, based on the ordering induced on web pages by the *Rank* vector. By looking at the ordering induced on specific query results, we have shown that even as few as 10 iterations can provide a useful ranking assignment. Given the timing results of Section 3.3, this convergence rate implies that we can compute a useful PageRank vector on a modestly equipped machine in roughly an hour, demonstrating the feasibility of client-side computation of personalized PageRank.

Acknowledgements

I would like to thank Professor Jeff Ullman and Professor Rajeev Motwani, as well as the rest of the Stanford MIDAS group, for invaluable discussions regarding the ideas in this paper. I would also like to thank the Stanford WebBase team for access to the web crawl data.

References

[1] Search Engine Watch: Up-to-date information on leading search engines. Located at http://www.searchenginewatch.com/.

[2] The Google Search Engine: Commercial search engine founded by the originators of PageRank. Located at http://www.google.com/.

[3] K. Bharat and A. Broder. A technique for measuring the relative size and overlap of public web search engines. In *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[4] S. Brin, R. Motwani, L. Page, and T. Winograd. What can you do with a web in your pocket. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 1998.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[6] S. Chakrabarti, B. Dom, D. Gibson, J. Kleinberg, P. Raghavan, and S. Rajagopalan. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[7] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proceedings of the Eighth International World Wide Web Conference*, 1999.

[8] J. Dean and M. Henzinger. Finding related pages in the world wide web. In *Proceedings of the Eighth International World Wide Web Conference*, 1999.

[9] E.-J. Im and K. Yelick. Optimizing sparse matrix vector multiplication on smps. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[10] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[11] S. Lawrence and C. L. Giles. Accessibility of information on the web. *Nature*, 400:107–109, July 1999.

[12] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, United Kingdom, 1995.

[13] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1998.

[14] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. In *IBM Journal of Research and Development*, volume 41. IBM, 1997.