Polytechnic UNIVERSITY
Brooklyn · Long Island · Westchester

I/O-Efficient Techniques for Computing Pagerank

Yen-Yu Chen Qingqing Gan Torsten Suel

POLYTECHNIC UNIVERSITY
VIRTUS VICTRIX FORTUNAE
HOMO ET HOMINIS OPERA PARTES NATURAE
1854

Department of Computer and Information Science

Technical Report
TR-CIS-2002-03
11/08/2002

I/O-Efficient Techniques for Computing Pagerank

Yen-Yu Chen *Qingqing Gan* *Torsten Suel*

CIS Department
Polytechnic University
Brooklyn, NY 11201
{yenyu, qq_gan, suel}@photon.poly.edu

Abstract

Over the last few years, most major search engines have integrated link-based ranking techniques in order to provide more accurate search results. One widely known approach is the Pagerank technique, which forms the basis of the Google ranking scheme, and which assigns a global importance measure to each page based on the importance of other pages pointing to it. The main advantage of the Pagerank measure is that it is independent of the query posed by a user; this means that it can be precomputed and then used to optimize the layout of the inverted index structure accordingly. However, computing the Pagerank measure requires implementing an iterative process on a massive graph corresponding to hundreds of millions of web pages and billions of hyperlinks.

In this paper, we study I/O-efficient techniques to perform this iterative computation. We derive two algorithms for Pagerank based on techniques proposed for out-of-core graph algorithms, and compare them to two existing algorithms proposed by Haveliwala. We also consider the implementation of a recently proposed topic-sensitive version of Pagerank. Our experimental results show that for very large data sets, significant improvements over previous results can be achieved on machines with moderate amounts of memory. On the other hand, at most minor improvements are possible on data sets that are only moderately larger than memory, which is the case in many practical scenarios.

A shorter version of this paper appears in the Proceedings of the Eleventh International Conference on Information and Knowledge Management (CIKM), November 2002.

1 Introduction

The World Wide Web has grown from a few thousand pages in 1993 to more than two billion pages at present. Due to this explosion in size, web search engines are becoming increasingly important as the primary means of locating relevant information. Given the large number of web pages on popular topics, one of the main challenges for a search engine is to provide a good ranking function that can efficiently identify the most useful results from among the many relevant pages. Most current engines perform ranking through a combination of term-based (or simply Boolean) query evaluation techniques, link-based techniques, and possibly user feedback (plus a healthy dose of preprocessing to fight search engine spam). We refer to [3, 11] for an overview of search engines architectures and ranking techniques, and to [8, 42] for background on information retrieval.

A significant amount of research has recently focused on link-based ranking techniques, i.e., techniques that use the hyperlink (or graph) structure of the web to identify interesting pages or relationships between pages. One such technique is the *Pagerank* technique underlying the Google search engine [11], which assigns a global importance measure to each page on the web based on the number and importance of other pages linking to it. Another basic approach, introduced by the HITS algorithm of Kleinberg [27] and subsequently modified and extended, e.g., in [5, 13, 30, 43], first identifies pages of interest through term-based techniques and then performs an analysis of only the graph neighborhood of these pages. The success of such link-based ranking techniques has also motivated a large amount of research focusing on the basic structure of the web [12], efficient computation with massive web graphs [9, 37, 40], and other applications of link-based techniques such as finding related pages [10], classifying pages [14], crawling important pages [19] or pages on a particular topic [16], or web data mining [28, 29], to name just a few.

Both Pagerank and HITS are based on an iterative process defined on the web graph (or a closely related graph) that assigns higher scores to pages that are more "central". One primary difference is that HITS is run at query time; this has the advantage of allowing the process to be tuned towards a particular query, e.g., by incorporating term-based techniques. The main drawback is the significant overhead in performing an iterative process for each of the thousands of queries per second that are submitted to the major engines. Pagerank, on the other hand, is independent of the query posed by a user; this means that it can be precomputed and then used to optimize the layout of the inverted index structure accordingly.

However, precomputing the Pagerank measure requires performing an iterative process on a massive graph corresponding to hundreds of millions of web pages and billions of hyperlinks. Such a large graph will not fit into the main memory of most machines if we use standard graph data structures, and there are two approaches to overcoming this problem. The first approach is to try to fit the graph structure into main memory by using compression techniques for the web graph proposed in [9, 37, 40]; this results in very fast running times but still requires substantial amounts of memory. The second approach, taken by Haveliwala [23], implements the Pagerank computation in an I/O-efficient manner through a sequence of scan operations on data files containing the graph structure and the intermediate values of the computation. In principle, this approach can work even if the data is significantly larger than the available memory.

In this paper, we follow this second approach and study techniques for the I/O-efficient computation of Pagerank in the case where the graph structure is significantly larger than main memory. We derive new algorithms for Pagerank based on techniques proposed for out-of-core graph algorithms, and compare them to two existing algorithms proposed by Haveliwala. We also consider the implementation of a recently proposed topic-sensitive version of Pagerank [24], which greatly increases the amount of processing that needs to be performed. Our results show that significant improvements over previous results can be achieved in certain cases.

The remainder of this paper is organized as follows. The next section describes the Pagerank technique in detail. Section 3 discusses related work, and Section 4 gives an overview of our results. Section 5 provides a detailed description of the previous and new algorithms for Pagerank that we study. Section 6 presents and discusses our experimental results. Finally, Section 7 offers some open problems and concluding remarks.

2 Fundamentals

We now review the Pagerank technique proposed in [11] in more detail. We will not yet discuss how to implement the computation in an I/O-efficient way; this is done in Section 5. We also describe an extension of Pagerank called *Topic Sensitive Pagerank* recently proposed by Haveliwala [24].

The Basic Idea: The basic goal in Pagerank is to assign to each page p on the web a global importance measure called its *rank value* $r(p)$. This rank value is itself determined by the rank values of all pages pointing to p, or more precisely

$$r(p) = \sum_{q \to p} \frac{r(q)}{d(q)}, \tag{1}$$

where $d(q)$ is the out-degree (number of outgoing hyperlinks) of page q. Writing the resulting system of equations in matrix form, we obtain

$$\vec{r} = A \cdot \vec{r},$$

where $\vec{r}$ is the vector of rank values over all pages, and A is a "degree-scaled" version of the adjacency matrix of the graph. Thus, the problem is essentially that of finding an eigenvector of A. As proposed in [11], this can be done by performing a simple and well-known iterative process that initializes the rank value of every page p (say, to $r^{(0)}(p) = 1/n$ where n is the total number of pages) and then iteratively computes

$$r^{(i)}(p) = \sum_{q \to p} \frac{r^{(i-1)}(q)}{d(q)}, \tag{2}$$

for $i = 1, 2, \ldots$. This is the basic process that is commonly implemented to compute Pagerank. (We also refer to [4] for a more detailed discussion of alternative formulations and iterative processes.)

Some Technical Issues: The above description of Pagerank ignores some important technical details. In general, the web graph is not strongly connected, leading to the following two problems:

- *Rank leaks* are pages with out-degree zero, which result in a loss of total rank value from the system under the above iterative process.

- *Rank sinks* are groups of pages forming a strongly connected component (e.g., a small cycle) with links entering the component but no links leaving it. These rank sinks act as sinks that accumulate large amounts of rank value.

Apart from complicating the underlying mathematics, both of these problems are also clearly undesirable from a ranking point of view. For example, rank sinks would result in artificially inflated ranks, and any outside linking would seriously reduce rank. (Some of these problems persist in the modified versions of Pagerank described next, though to a lesser degree.) The solution to the first problem is to repeatedly prune the input graph such that all or most rank leaks are removed[1].

To resolve the issue of rank sinks, we appeal to the *random surfer* model of Pagerank [11], which observes that the value $r^{(i)}(p)$ computed by Equation (2) is proportional to the probability of being at page p after starting at a random page and following random outgoing links for i steps. Note that a rank sink will trap such a surfer forever. The solution taken by Pagerank is to add random jumps to the iterative process, as follows: with some probability α, the surfer follows a random outgoing link; otherwise, the surfer jumps to a random page in the graph. Typical values for α are between 0.8 and 0.9; thus, after a moderate number of iterations, the surfer is likely to jump out of a rank sink. (In addition, the parameter α also has the effect of dampening peaks in the rank values, and of limiting the influence of nodes to their local neighborhoods; a discussion of these issues is beyond the scope of this paper.)

This leads to the following modification of Equation (2), which is equivalent to the modified random surfer model if all leaks have been removed:

$$r^{(i)}(p) = (1-\alpha)\frac{R^{(0)}}{n} + \alpha \cdot \sum_{q \to p} \frac{r^{(i-1)}(q)}{d(q)}, \tag{3}$$

where n is the total number of pages and $R^{(0)} = \sum_p r^{(0)}(p)$ is the amount of rank initially inserted into the network. This is the iterative process for Pagerank that we assume in the remainder of the paper. As discussed in [23, 4], it is usually sufficient to run the process for 20 to 50 iterations, after which the relative ordering of pages is close to that of the converged state. In this paper, we are not concerned with this rate of convergence, but instead focus on optimizing the time needed per iteration. We describe the I/O-efficient implementation of this process in Section 5, and our data set and preprocessing steps in Section 6.

Topic-Sensitive Pagerank: Recall that Pagerank assigns ranks to pages independent of the topic of a query, thus allowing efficient preprocessing of the data set. On the other hand, it has been argued that a link-based ranking scheme that takes the topic of a query into account might be able to return better results than a query-independent scheme [13, 27]. The *Topic-Sensitive Pagerank* approach recently proposed by Haveliwala [24] attempts to combine the advantages of both approaches. The idea is to perform multiple Pagerank computations, each biased towards a particular topic taken from a standard hierarchy such as Yahoo or the Open Directory Project, and to then compute a ranking for a particular query by combining several precomputed rankings from topics that are relevant to the query.

In this paper, we are interested only in the efficient precomputation of multiple *Topic-Sensitive Pagerank* vectors, and do not consider the issue of how to use these vectors in query processing. The algorithm for performing a single *Topic-Sensitive Pagerank* computation is as follows:

- Select a limited number of *special pages* that are highly relevant to the chosen topic, e.g., by choosing pages listed under the corresponding category of the Open Directory Project.

[1]Alternatively, leak nodes can be handled by adding a corrective term that reinserts leaked rank back into the system by distributing it evenly over all nodes.

- Modify the random surfer model so that instead of jumping to a random page with probability $(1-\alpha)$, we perform a jump to a random page among the special pages.

This has the effect of modifying our recurrence to the following:

$$r^{(i)}(p) = \begin{cases} (1-\alpha)\frac{R^{(0)}}{s} + \alpha \cdot \sum_{q \to p} \frac{r^{(i-1)}(q)}{d(q)} & \text{if } p \text{ is a special page, and} \\ \alpha \cdot \sum_{q \to p} \frac{r^{(i-1)}(q)}{d(q)} & \text{otherwise,} \end{cases} \tag{4}$$

where s is the number of special pages. As we discuss later, we can modify our algorithms for Pagerank to simultaneously compute multiple topic-sensitive rankings, resulting in a computation on a new graph with proportionally more pages that is often more efficient then a separate computation of each ranking.

3 Discussion of Related Work

As mentioned in the beginning, there is a large amount of recent academic and industrial work on web search technology. Most closely related to our work is the also quite extensive literature on the analysis and utilization of the hyperlink structure of the web; see [3, 15, 25] for an overview. In general, link-based ranking is closely related to other applications of link analysis, such as clustering, categorization, or data mining, although a discussion of these issues is beyond the scope of this paper.

Various heuristics for link-based ranking, such as counting in-degrees of nodes or computing simple graph measures, have been known since the early days of web search engines [31, 36]. The Pagerank approach was introduced in 1998 as the basis of the Google ranking scheme [11, 34]. Another related approach to link-based ranking, proposed at around the same time, is the HITS algorithm of Kleinberg [27]. Over the last few years, numerous extensions and variations of these basic approaches have been proposed; see [5, 13, 24, 30, 43] for a few examples. We focus here on the implementation of the basic Pagerank algorithm on massive graphs, and on its topic-sensitive extension in [24]. We note that many of the extensions of the HITS approach are run on small subsets of the graph, often at query time, and thus do not fit into the framework in this paper[2].

The initial papers on Pagerank [11, 34] did not discuss the efficient implementation of the algorithm in detail. The I/O-efficient implementation of Pagerank was subsequently studied by Haveliwala [23], who proposes an algorithm based on techniques from databases and linear algebra. This algorithm (which we will refer to as *Haveliwala's Algorithm*, is then compared to a simpler scheme that assumes that the rank vector for the entire graph fits into main memory (referred to as the *Naive Algorithm*). While the results show that *Haveliwala's Algorithm* is quite efficient in many cases, we show in this paper that additional improvements are possible for very large graphs and for the case of *Topic-Sensitive Pagerank.*

We note that an alternative approach is to compute Pagerank completely in main memory using a highly compressed representation of the web graph [2, 9, 22, 37, 40]. In this case, enough main memory is required to store the rank vector for the entire graph plus the compressed representation of the hyperlink structure. The currently best compression scheme requires about 0.6 bytes per

[2]One challenge faced by the HITS approach is how to retrieve the necessary graph and meta data at query time, which requires efficient access structures for lookups, as opposed to the off-line out-of-core methods that we study.

hyperlink [37], resulting in a total space requirement of about $2 + 4.2 = 6.2$ GB for a graph with 500 million nodes and 7 billion edges. In contrast, our techniques are efficient even for machines with very moderate amounts of main-memory. Of course, it can be argued that an 8 GB machine is now well within the reach of most organizations involved in large-scale web research; we discuss this issue again in the next subsection. (Alternatively, the graph could also be partitioned over several nodes of a cluster, each containing a more moderate amount of memory.)

Very recent work by Arasu et al. [4] considers modifications of the basic iterative process in Pagerank that might lead to faster convergence. In particular, using a Gauss-Seidel iteration instead of the basic Jacobi iteration of Equation (3) appears to decrease the number of iterations needed by about a factor of 2. We note that Gauss-Seidel iteration would require modifications in both Haveliwala's and our algorithms that would likely increase the running time per iteration. There are a number of other well-known techniques [7, 41] from linear algebra that might improve convergence; note however that the potential improvement is limited given the modest number of iterations required even by the basic iterative process. Finally, recent work in [18] proposes to reduce the cost of computing Pagerank by incrementally recomputing rank values as the link structure evolves.

4 Contributions of this Paper

In this paper, we study techniques for the I/O-efficient implementation of Pagerank in the case where the graph is too large to fit in main memory. We describe two new algorithms, and compare them to those proposed by Haveliwala in [23]. In particular,

(1) We describe an algorithm based on the very general paradigm for I/O-efficient graph algorithms proposed by Chiang et al. in [17]. The algorithm, called the *Sort-Merge Algorithm*, is conceptually very simple, and is interesting to contrast to *Haveliwala's Algorithm* as both are closely related to different join algorithms in databases. As we show, this algorithm is however not very good in practice, and hence primarily of conceptual interest.

(2) We propose another algorithm, called the *Split-Accumulate Algorithm*, that overcomes the bad scaling behavior of *Haveliwala's Algorithm* on very large data sets, while avoiding the overhead of the *Sort-Merge Algorithm*. In fact, the algorithm is somewhat reminiscent of hash-based join algorithms in databases, and it exploits the locality inherent in the link structure of the web.

(3) We perform an experimental evaluation of the two new algorithms and the two described in [23] using a large graph from a recent web crawl. The evaluation shows significant improvements over previous approaches for the *Split-Accumulate Algorithm* if the graphs are large compared to the amount of memory. On the other hand, at most minor improvements are possible on data sets that are only moderately larger than memory, which is the case in many practical scenarios.

(4) We study the efficient implementation of the recently proposed *Topic-Sensitive Pagerank* [24].

Motivation: We now briefly discuss the motivation for our work. As mentioned earlier, it can be argued that even for fairly large graphs we could perform the Pagerank computation completely in memory given access to a machine with 8 GB or more of main memory and a highly optimized compression scheme for graphs. Moreover, even if the link structure of the graph does not fit in memory, the *Naive Algorithm* in [23] will provide an efficient solution as long as we can store one floating point number for each node of the graph. We believe that our results are nonetheless of interest because of the following:

- The new algorithms provide significant benefits for machines with more moderate amounts of main memory. As discussed by Haveliwala [23], this might be the case if Pagerank is performed on a client machine rather than at the search engine company, in which case the computation would also have to share resources with other tasks in the system. We believe that search and in particular ranking will become increasingly client-based in the future (although it is not clear that Pagerank would be the method of choice for query-time link analysis).
- The situation is somewhat different for *Topic-Sensitive Pagerank* where the space requirement of the rank vector in the *Naive Algorithm* increases from $O(n)$ to $O(nk)$, where k is the number of topics (unless we are willing to perform k distinct Pagerank computations at a significant cost in time). In this case, I/O-efficient techniques have significant benefits over the *Naive Algorithm*. (We note however that for larger numbers of topics, it might be interesting to develop pruning techniques that limit the computation for each topic to a subset of the graph. We are not aware of published work on this issue.)
- We believe that the techniques that we study are of more general interest for computations on massive graphs, which have applications in several other areas besides web search [20]. While a number of theoretical results on massive graphs have been published, there are still relatively few experimental studies on real data. As we observe, some of the theoretically good approaches in the literature [17] are not optimal for graphs that are only moderately larger than memory, and in fact our *Split-Accumulate Algorithm* attempts to overcome this issue. We note that iterative processes very similar to Pagerank have, e.g., been proposed for various multi-commodity flow problems [6, 26, 32], which could be solved in an I/O-efficient way using the same techniques. Conversely, techniques based on network flow have also recently been used in the link-based analysis of web pages [21].

5 Algorithms

We now describe the different algorithms that we study in this paper. We begin with the two algorithms described by Haveliwala [23] and then introduce the two new ones that we propose. The modifications required to the algorithms to compute *Topic-Sensitive Pagerank* are discussed in the relevant subsection of the experimental evaluation.

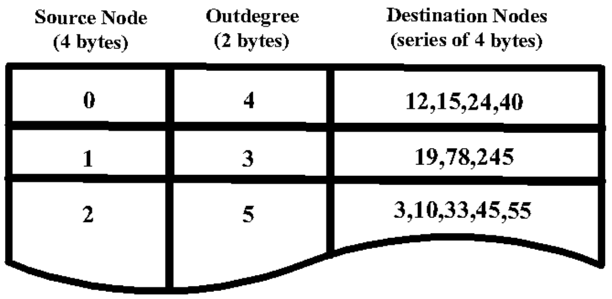
Assume that the input data exists in the form of two files, a *URL file* in text format containing the URLs of all pages in the graph in alphabetical order, and a binary *link file* containing all edges in arbitrary order, where each edge is a pair of integers referring to line numbers in the URL file. This format is obtained through a sequence of preprocessing steps as described in Subsection 6.1.

The Pagerank algorithms do not access the URL file at all, and in fact each of them expects a slightly different format for the link file, as described below. Also recall from Equation (3) that each iteration of the computation can be stated as a vector-matrix multiplication between a vector V containing the rank values of the nodes before the iteration (also called *source*) and a matrix implied by the link structure, resulting in a vector V' of new rank values after the iteration (also called *destination*).

In the descriptions, we state the cost of each algorithm in terms of the total amount of data read and written in each iteration. Note that all disk I/O is performed in blocks of size at least $512KB$, and that except for the internal sort and the heap-based merge in the *Sort-Merge Algorithm*, no extensive internal computations are performed. Thus, this measure provides a reasonable first estimate of the actual running time.

5.1 The Naive Algorithm

This first algorithm assumes that we can store one floating point number for each node in memory. The algorithm uses a binary link file L illustrated in Figure 1. The link file contains one record for each node, identifying the node, its out-degree, and all destination nodes linked by it; the file is sorted by the line number of the source node. In addition, we have two vectors of 32-bit floating point numbers, a source vector on disk containing the rank values before the iteration, and a destination vector in memory containing the values afterwards.



Source Node (4 bytes)	Outdegree (2 bytes)	Destination Nodes (series of 4 bytes)
0	4	12,15,24,40
1	3	19,78,245
2	5	3,10,33,45,55

Figure 1 : Link Structure File L

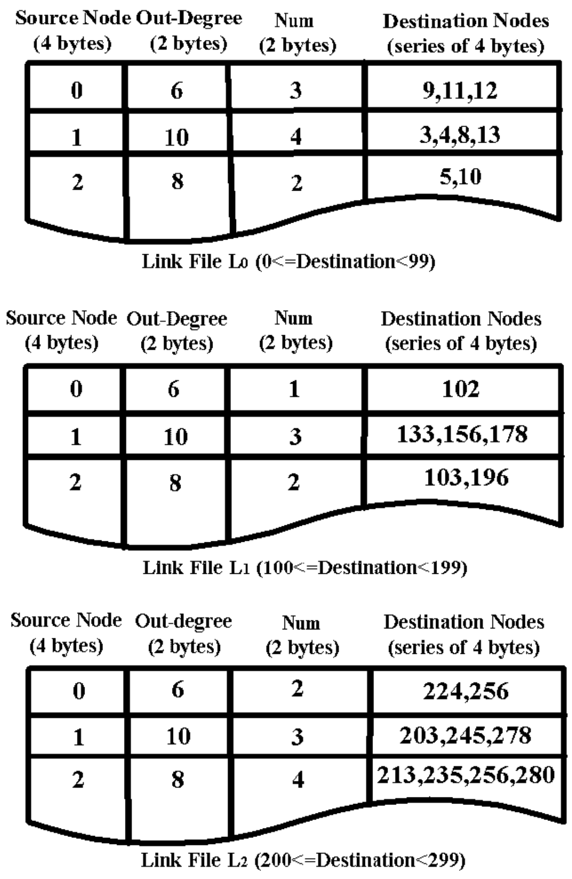
In each iteration, we first initialize the destination vector to $(1-\alpha)\frac{R^{(0)}}{n}$. Since both source and link file are sorted by the line numbers of the source nodes, we can scan over both files in a synchronized fashion, adding for each source and each link some additional rank value to the appropriate destination in the memory-resident destination vector. Finally, we write out the destination vector to be used as source in the next iteration. Thus, assuming memory is large enough to hold the destination vector, the total I/O cost for this strategy is given by

$$C_{naive} = |V| + |L| + |V'| = 2 \cdot |V| + |L|,$$

where $|L|$ is typically by a factor of 5 to 15 larger than $|V|$. However, if the main memory is not large enough, then the time will be far larger due to disk swapping.

5.2 Haveliwala's Algorithm

We now review the improved scheme proposed in [23]. The idea is to partition the destination vector into d blocks V'_i that each fit into main memory, and to compute one block at a time. However, it would be quite expensive to repeatedly scan the large link file in this process. To avoid this, we preprocess the link file and partition it into d files L_i, as illustrated in Figure 2, where L_i only contains the links pointing to nodes in block V'_i of the destination vector, sorted by source node as before.



Source Node (4 bytes)	Out-Degree (2 bytes)	Num (2 bytes)	Destination Nodes (series of 4 bytes)
0	6	3	9,11,12
1	10	4	3,4,8,13
2	8	2	5,10

Link File L0 (0<=Destination<99)

Source Node (4 bytes)	Out-Degree (2 bytes)	Num (2 bytes)	Destination Nodes (series of 4 bytes)
0	6	1	102
1	10	3	133,156,178
2	8	2	103,196

Link File L1 (100<=Destination<199)

Source Node (4 bytes)	Out-degree (2 bytes)	Num (2 bytes)	Destination Nodes (series of 4 bytes)
0	6	2	224,256
1	10	3	203,245,278
2	8	4	213,235,256,280

Link File L2 (200<=Destination<299)

Figure 2 : Partitioned Link Files (first 3 buckets)

Thus, before the first iteration we have to perform a preprocessing step to create the L_i. In each iteration, we then essentially run the *Naive Algorithm* on V and L_i to compute block V'_i of the destination vector, for $0 \leq i < d$. Thus, the cost of each iteration is:

$$C_{haveli} = d \cdot |V| + \sum_{0 \leq i < d} |V'_i| + \sum_{0 \leq i < d} |L_i| = (d+1) \cdot |V| + (1+\epsilon) \cdot |L|,$$

where the term ϵ takes into account the slightly less compact format for the L_i [23]. Note that the source file has to be read d times in this scheme; this limits its scalability to massive data sets as the number d of partitions increases. In addition, ϵ also increases slowly with d; for moderate values of d we have $\epsilon \approx 0.1$ while the maximum possible (but unlikely) value is $2 - \frac{6}{d+2}$. As observed in [23], this algorithm is quite similar in structure to the Block Nested-Loop Join algorithm in database systems. which also performs very well for data sets of moderate size but eventually loses out to more scalable approaches.

5.3 The Sort-Merge Algorithm

This algorithm is based on the theoretical framework for out-of-core graph algorithms introduced by Chiang et al. [17]. The basic idea is to partition the computation into repeated sorting steps. Note that in each iteration of Pagerank, we essentially have to transmit an amount of rank value from the source to the destination of each link. Using the framework in [17], this can be achieved by creating for each link a *packet* that contains the line number of the destination and an amount of rank value that has to be transmitted to that destination. We can then *route* these 8-byte packets by sorting them by destination and combining the ranks into the destination node. We note that this approach is reminiscent of packet routing problems in parallel machines, and that the resulting algorithm is also quite similar to the Sort-Merge Join algorithm in database systems.

A naive implementation of this would first perform a *synchronized scan* over V and the link file L from the *Naive Algorithm* to create the packets, i.e., a scan over V and L such that matching records from both files are processed simultaneously. These packets would then have to be written out to disk, sorted by destination, and then scanned again to create the destination vector V'. We can improve on this by buffering as many newly created packets as possible in memory and writing them out in a sorted run. This way, we can immediately perform the merge of these sorted runs of packets in the next phase. From the output of the merge we can directly create the destination vector without writing the packets out again. The I/O cost of this approach is thus given by $C_{sort} = |V| + |V'| + |L| + 2 \cdot |P|$ where $|P|$ is the total size of the generated packets that need to be written in and out once. Note that in the first iteration, we can directly create the packets from the initialization values.

There are several further optimizations that we considered. First, after sorting and before writing out the packets, we can combine packets with the same destination. As it turns out, this significantly decreases the total number of packets, due to the large degree of locality in the web graph (see, e.g., [12, 37, 40] for discussions of this issue). Second, instead of directly writing out the combined packets, we can now continue to generate packets and place them in the freed-up part of the output buffer. When the buffer is full, we can again sort and combine packets, and repeat this process. We determined that performance was optimized by repeating this process 6 times, sorting each time only the newly generated packets, except for the last time when we applied sorting and combining to the entire buffer. As a result, the size $|P|$ of the generated packets drops from slightly larger than $|L|$ to only a fraction of $|L|$.

There are two further optimizations that we did not include in our implementation. Note that we write out the result V of an iteration, only to read it in again immediately afterwards in the next iteration. Instead, we could entirely get rid of the source and destination vector files on disk, and directly create new packets for the next iteration as we process the output of the merge from the previous iteration. This would remove the term $|V| + |V'|$ from the cost, resulting in savings of at most $5 - 10\%$ but complicating the code. Finally, we briefly experimented with a simple encoding scheme for packet destinations that uses a one-byte offset from the previous destination in most cases. However, an implementation of this scheme in the context of the *Split-Accumulate*

Algorithm actually resulted in a slight slowdown due to the need for byte-level operations, and hence we decided not to delve any deeper into such encoding techniques.

5.4 The Split-Accumulate Algorithm

The second algorithm we propose combines some of the properties of *Haveliwala's Algorithm* and the *Sort-Merge Algorithm*. In particular, it uses a set of link structure files $\overline{L_i}$ very similar of the L_i in *Haveliwala's Algorithm*. Instead of performing a local sort followed by a merge as in the *Sort-Merge Algorithm*, it performs a split into buckets followed by an aggregation in a table (which could also be interpreted as a counting-based sort).

The algorithm splits the source vector into d blocks V_i, such that the 4-byte rank values of all nodes in a block fit into memory. The blocks V_i only exist in main-memory, and the algorithm uses three sets of files $\overline{L_i}$, $\overline{P_i}$, and O_i, $0 \leq i < d$. File $\overline{L_i}$ contains information on all links with source node in block V_i, sorted by destination, as shown in Figure 3. Note that this is essentially the reverse of the files L_i in *Haveliwala's Algorithm* except that the out-degrees of all sources in V_i are not stored in $\overline{L_i}$ but in a separate file O_i that is a vector of 2-byte integers. File $\overline{P_i}$ is defined as containing all packets of rank values with destination in block V_i, in arbitrary order.

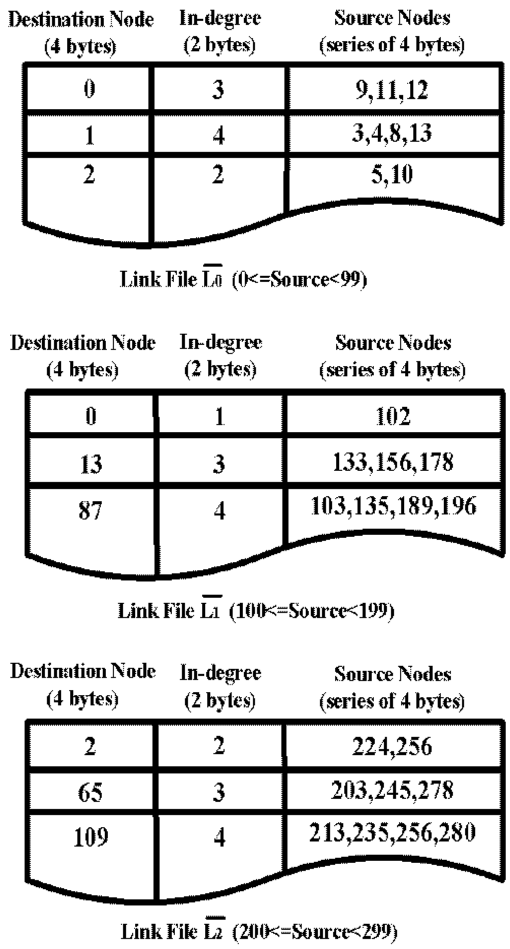
In each iteration of the algorithm, we proceed in d phases from $i = 0$ to $d - 1$. In the ith phase, we first initialize all values in block V_i in memory to $(1-\alpha)\frac{R^{(0)}}{n}$. We then scan the file $\overline{P_i}$ of packets with destinations in V_i, and add the rank values in each packet to the appropriate entry in V_i. After $\overline{P_i}$ has been completely processed, we scan the file O_i of out-degrees, and divide each rank value in V_i by its out-degree. We now start reading $\overline{L_i}$, and for each record in $\overline{L_i}$ consisting of several sources in V_i and a destination in V_j, we write one packet with this destination node and the total amount of rank to be transmitted to it from these sources into output file $\overline{P'_j}$ (which will become file $\overline{P_j}$ in the next iteration).

The processing of one iteration is illustrated in Figure 4 for $d = 3$. We first read in $\overline{P_0}$, then perform a sort of vector-matrix multiplication with the file $\overline{L_0}$ that is scanned. This creates output into all three output files $\overline{P'_i}$, but due to link locality most output is generated for file $\overline{P'_0}$. Then we repeat this process for $\overline{P_1}$ and $\overline{L_1}$, resulting in most output being sent to $\overline{P'_1}$, and finally for $\overline{P_2}$ and $\overline{L_2}$.

As in the *Sort-Merge Algorithm*, we can combine packets with the same destination. In fact, combining packets is simpler and more efficient in this case, since all packets originating from the same file $\overline{P_i}$ are written in sorted order into $\overline{P'_j}$. Thus, no in-memory sorting of packets is needed since we can just compare the new destination to that of the packet previously written to the same output file $\overline{P'_j}$. (As mentioned before, a 1-byte offset encoding of the destinations did not result in any further improvements.) The I/O cost of this algorithm is

$$C_{split} = \sum_{0 \leq i < d} \left(|O_i| + |\overline{L_i}| + 2 \cdot |\overline{P_i}|\right) = 0.5 \cdot |V| + (1+\epsilon')|L| + 2 \cdot |\overline{P}| \approx (1+\epsilon)|L| + 2 \cdot |\overline{P}|,$$

where the factor $(1+\epsilon')$ models the slightly less concise representation of the link files $\overline{L_i}$ as compared to L in the *Naive Algorithm*. The term $0.5 \cdot |V| + (1+\epsilon')|L|$ for the O_i and $\overline{L_i}$ files is about the same size as the term $(1+\epsilon)|L|$ for the L_i in *Haveliwala's Algorithm* (not surprisingly given the similarity of their structure and content). Moreover, the total size $|\overline{P}|$ of the packet files



Destination Node (4 bytes)	In-degree (2 bytes)	Source Nodes (series of 4 bytes)
0	3	9,11,12
1	4	3,4,8,13
2	2	5,10

Link File $\overline{L_0}$ (0<=Source<99)

Destination Node (4 bytes)	In-degree (2 bytes)	Source Nodes (series of 4 bytes)
0	1	102
13	3	133,156,178
87	4	103,135,189,196

Link File $\overline{L_1}$ (100<=Source<199)

Destination Node (4 bytes)	In-degree (2 bytes)	Source Nodes (series of 4 bytes)
2	2	224,256
65	3	203,245,278
109	4	213,235,256,280

Link File $\overline{L_2}$ (200<=Source<299)

Figure 3: Reverse Partitioned Link Files (first 3 buckets)

is significantly smaller than $|L|$, and in fact is typically less than $3 \cdot |V|$, due to the effects of combining.

We note that the *Split-Accumulate Algorithm* is in fact somewhat reminiscent of the hash-based join algorithms widely used in database systems. Compared to the *Sort-Merge Algorithm,* instead of sorting and merging we are now splitting the packets into different buckets $\overline{P_i}$ by destination, and then directly accumulating rank values at the destination using a table. This has several advantages: it removes the computational overhead of the internal sort and the heap-based merge, it automatically combines several links into one packet, and it results in slightly smaller (and fewer) output files $\overline{P_i'}$.

6 Experimental Results

We now present the experimental evaluation of the different algorithms. We first describe the machine setup, data sets, and preprocessing steps. Subsection 6.2 to 6.4 present the experimental results for Pagerank, and Subsection 6.5 presents the results for *Topic-Sensitive Pagerank.*

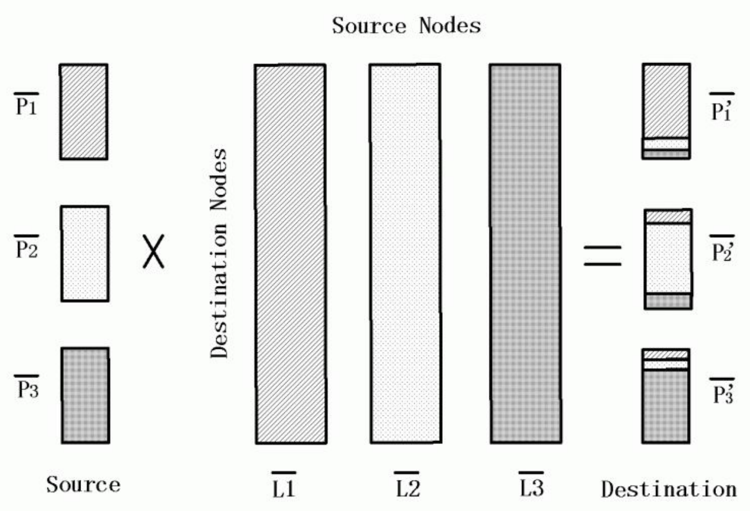


Figure 4: Matrix Multiplication

6.1 Experimental Setup

Hardware: For all experiments, we used a Sun Blade 100 workstation running Solaris 8 with a 500 Mhz UltraSparc IIe CPU and two 100 GB, 7200 RPM Western Digital WD1000BB hard disks. Only one disk was used for the experiments, and the disk was less than 60% full during all except the largest experiments. All algorithms were coded in `C++` using the `fread`, `read`, `fwrite` and `write` operations provided by the standard libraries, with all disk accesses being of size $512KB$ or larger.

Memory Consumption: We considered machine configurations from $32MB$ to $1GB$ of main memory, and for each configuration we optimized the parameters of the algorithms to minimize execution time. This involves choosing the best value of d for the *Split-Accumulate Algorithm* and *Haveliwala's Algorithm*, and choosing the output buffer size of the *Sort-Merge Algorithm*.

We physically modified the amount of memory from $128MB$ to $1GB$. The runs for $32MB$ and $64MB$ were performed on a machine with $128MB$ of physical memory, but using parameter settings for $32MB$ and $64MB$. One potential problem with this is that the additional physical memory could be used by the OS for buffering purposes, resulting in unrealistic running times. To check for this, we ran tests on several larger physical configurations and did not observe significant differences due to extra memory available for caching. (The reason is probably that the algorithms are based on scans of very large input files that do not benefit from LRU-type caching approaches.)

Input Data: Our input data was derived from a crawl of 120 million web pages (including more than 100 million distinct pages) performed by the PolyBot web crawler [39] in May of 2001. The crawl started at a hundred homepages of US Universities, and was performed in a breadth-first manner[3]. As observed in [33], such a crawl will find most pages with significant Pagerank value. The total size of the data was around $1.8TB$, and the link data was extracted and converted into the

[3] Subject to a 30 second interval between accesses to the same server that may cause reordering from strict breadth-first.

URL file and *link file* format described in Section 5 through a series of I/O-efficient (but not overly optimized) parsing, sorting, and merging operations on a cluster of Linux and Solaris machines. We note that the preprocessing of the data took several weeks of CPU time, and thus is much larger than the actual Pagerank computations that we report. We extracted a graph with 327 million URLs and 1.32 billion edges, including URLs not crawled but referenced by crawled pages. From this graph, we generated two types of input graphs.

Pruned Graphs: Following Haveliwala [23], we pruned the graph twice by removing nodes with no out-going links. This resulted in a highly connected graph with $44,792,052$ nodes and $665,901,912$ links, for an average out-degree of 14.5. We created additional artificial graphs of larger size by concatenating several copies of the above graph, and then connecting the copies by "rerouting" some of the edges into the other copies, subject to a randomized heuristic that aims to preserve locality, in-degree, and out-degree distributions. This created graphs of 2, 4, and 8 times the size of the base graph. We note that while there are formal random graph models that can be used to generate web graphs, see, e.g., [12], most of these models do not account for link locality, which is crucial for our experiments. Another problem with the formal models is that the graphs themselves need to be generated in an I/O-efficient manner given their size.

Graphs with Backlinks: Following the discussion in [35], we generated another input graph by keeping all nodes and edges, and adding backlinks from every node of out-degree zero to the nodes pointing to it. This generated a graph with 327 million URLs and 1.97 billion edges, for an average out-degree of 6.0.

Thus, decisions on whether to prune leaks or local and nepotistic links, or to add backlinks, can significantly impact out-degree and link locality. The first approach of pruning leaks but keeping all other links results in a smaller but highly connected graph with large out-degree and a high degree of locality in the link structure. The second approach results in a much larger graph of more moderate out-degree, We note that a smaller size and high out-degree is a significant disadvantage for our algorithms compared to *Haveliwala's Algorithm*. On the other hand, a high degree of locality in the link structure is an advantage, as it allows efficient combining of packets in our algorithms.

6.2 Results for Pruned Data

In our first set of experiments, we ran all four algorithms on the pruned web graph with 45 million nodes and 652 million edges. We used four different setups, corresponding to different amounts of available main memory. We report three different results for each experiment: (a) the running time in seconds, (b) the I/O throughput in megabytes per second, and (c) the running time predicted by the simple cost estimates in Section 5. The memory sizes of 32, 64, 128, and $256MB$ correspond to values of 6, 4, 3, and 1, respectively, for the number of partitions d in *Haveliwala's Algorithm* and the *Split-Accumulate Algorithm*.

Figure 5 contains the measured running times for the algorithms under four different memory configurations. We note that the *Naive Algorithm* requires at least $180MB$ of memory to hold the rank vector. As expected, the *Naive Algorithm* is the best choice when applicable (we did not attempt to time the algorithm when the vector does not fit, but the result would obviously be quite bad). Comparing *Haveliwala's Algorithm* and the *Split-Accumulate Algorithm*, we see that the latter does significantly better when memory size is much smaller than data size, while being

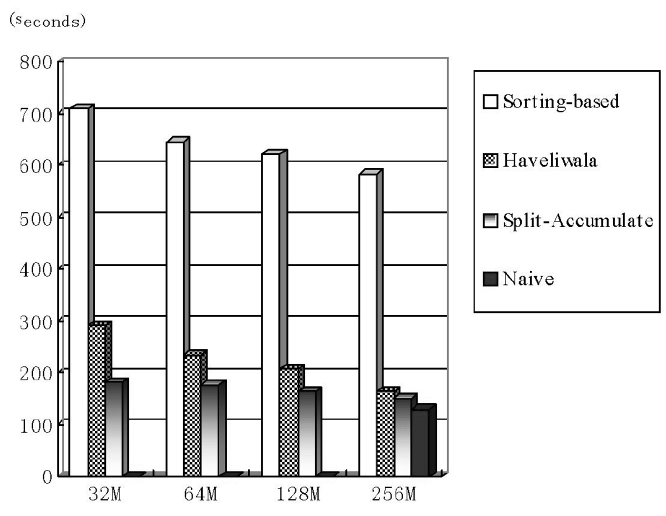


Figure 5: Running times in seconds for all four algorithms under different memory settings.

comparable in speed otherwise. Of course, a memory size of $32MB$ or $64MB$ is unrealistically small given that even the base version of the low-end Blade 100 we used comes with $128MB$ of memory.

The poor performance of the simple *Sort-Merge Algorithm* at first was quite disappointing to us. The primary reason for this performance is the overhead due to the internal sort of the packets in the output buffers. This step was implemented through a fairly optimized radix sort for 8-byte records with 32-bit keys that achieves a throughput of about $16MB/s$, comparable to disk speed[4]. However, this sorting step is applied to the packets before combining, and thus involves a much larger data set than the one that is actually written out afterwards. Moreover, the number of sorted runs created by the algorithm, which is proportional to $|P|/M$, is larger than the split factor d used in the *Sort-Accumulate Algorithm*, which is proportional to $|V|/M$, where M is the memory size. This creates additional overhead when the sorted runs are merged, necessitating a move to a two-level merge at a fairly early point (though not yet in the data set we use here). In summary, the *Sort-Merge Algorithm* performs significantly worse than the others, and we decided to drop it from subsequent experiments on larger graphs.

We now discuss the relationship of these results to the cost estimates established in Section 5. Table 6.1 shows the I/O throughput per second achieved by the different algorithms, which varies between 6 and $25MB/s$. The low throughput of the *Sort-Merge Algorithm* is due to the bottleneck in the internal sort. For the other three algorithms the rates are much closer. To separate the impacts of disk I/O and internal computations on running times, we observed that a simple `C` program that does not perform any calculations achieved a maximum I/O throughput with `fread` and `fwrite`

[4]We note that sorting performance on UltraSparc IIe processors is significantly worse than on state-of-the-art Intel processors, and that our 500 Mhz Blade 100 surprisingly did even worse on this than comparable Ultra 5 and 10 machines with slower processors.

Algorithm	32 MB	64 MB	128 MB	256 MB
Naive	-	-	-	25.09
Haveliwala	18.36	19.45	18.65	21.39
Split-Accumulate	20.69	21.06	20.58	23.75
Sort-Merge	6.01	6.53	6.73	7.04

Table 6.1: I/O throughput in MB/s for different algorithms and memory consumption levels

of about $25MB/s$ on this machine. A naive estimate[5] could be made that if the internal processing in the Pagerank algorithms can run at about $100MB/s$, then the overall throughput is $20MB/s$, about the rate observed in most runs.

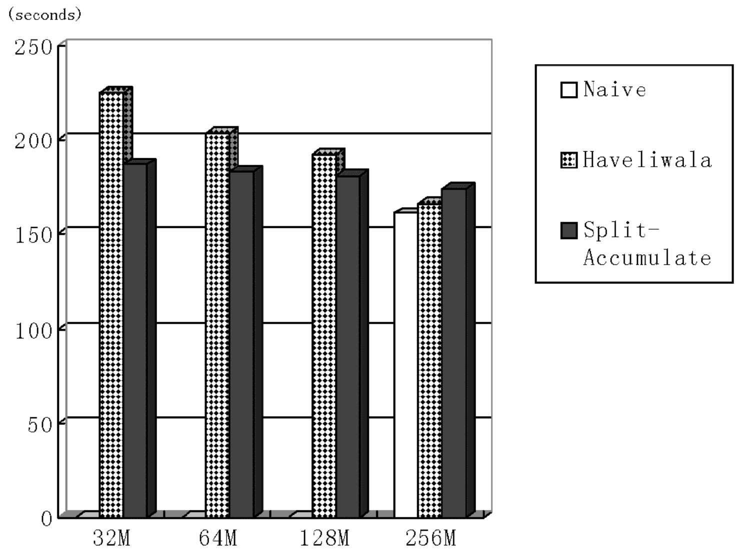


Figure 6: Predicted running times in seconds based on I/O cost estimates.

In Figure 6, we show the running times that we would obtain if all algorithms (except the *Sort-Merge Algorithm*) processed their data at such a uniform rate of $20MB/s$, using the cost estimates from Section 5. Note that we did not attempt to completely optimize the internal computation rates of the different algorithms; the estimates in the figure indicate that the influence of such optimizations on the relative performance of the algorithms is likely to be minor except for the case of the *Sort-Merge Algorithm*.

Finally, we look at how the split factor d and the combining techniques affected the sizes of

[5]Assuming that I/O and CPU work cannot be overlapped well on this platform, as indicated by our attempts at using asynchronous I/O.

	d = 6	d = 4	d = 3	d = 1
$\|L\|$	-	-	-	2878
$\|V\| = \sum \|V_i\|$	180	180	180	180
$\sum \|O_i\|$	90	90	90	90
$\sum \|L_i\|$	3241	3169	3131	2966
$\sum \|\overline{L_i}\|$	2902	2877	2864	2816
$\sum \|\overline{P_i}\|$	423	399	391	340
$\sum \|P_i\|$	510	490	472	433
Sort-Merge degree	51	30	19	8

Table 6.2: Sizes of data files in MB, and number of sorted runs, for different values of d.

the L_i, $\overline{L_i}$, P_i, and $\overline{P_i}$ files. Table 6.2 shows that the overhead in the L_i and $\overline{L_i}$ files, compared to the case of $d = 1$, is very moderate, particularly for the $\overline{L_i}$ due to the higher skew in the in-degree distribution that results in a slightly more compact representation. For the P_i and $\overline{P_i}$ files, we see that combining is quite effective even for $d = 6$, resulting in files that are significantly smaller than the link files. This is a major reason for the speed of the *Split-Accumulate Algorithm*. For the *Sort-Merge Algorithm*, we also see the relatively high degree in the merge phase; e.g., for $32MB$, 51 sorted runs had to be merged.

6.3 Results for Scaled Pruned Data

Due to the limited size of the real data, we had to assume fairly small amounts of memory in order to get interesting trade-offs in the performance. Next, we run experiments on larger data sets obtained by scaling the real data as described above.

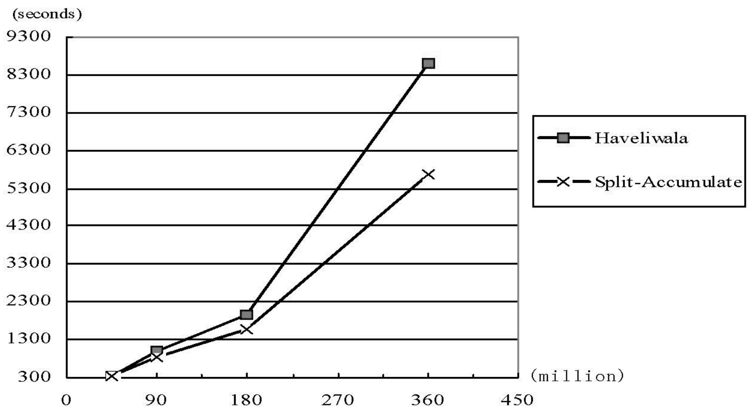


Figure 7: Running times in seconds versus data size for scaled pruned graphs.

Figure 7 shows the running times for the two algorithms on $256MB$ of memory as we scale the data set to twice and four times its original size, resulting in 90 million nodes and 1.306 billion links, 180 million nodes and 2.612 billion links, and 360 million nodes and 5.224 billion links, respectively. As expected, the advantage of the *Split-Accumulate Algorithm* becomes more pronounced, and we would expect additional gains over *Haveliwala's Algorithm* as we scale the data size further up. As shown in Figure 8, the advantage of the *Split-Accumulate Algorithm* over the others increases with smaller memory size, as we would expect.

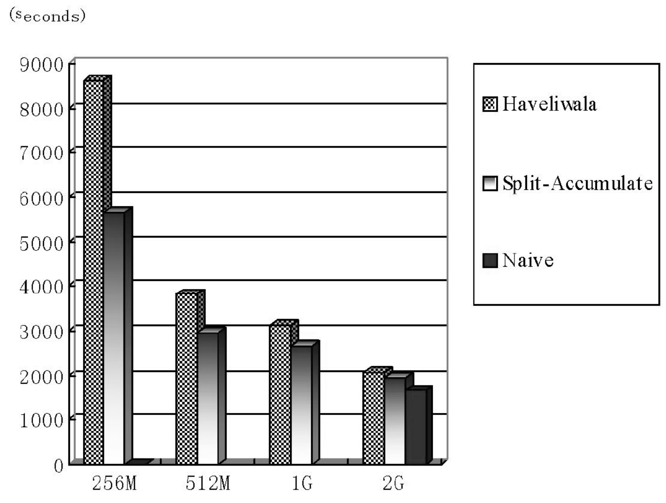


Figure 8: Running times on 360 million nodes for various memory sizes.

6.4 Results for Data with Backlinks

We also ran experiments for the case whether we do not prune leaks, but instead add links from each leak to all pages pointing to it, as suggested in [35]. The resulting graph has 327 million nodes and 1.97 billion edges. Note that in this case, the average out-degree is significantly smaller than in the pruned case.

The results for different memory configurations are shown in Figure 9. Not surprisingly, we see a larger difference in the running time between the *Split-Accumulate Algorithm* and *Haveliwala's Algorithm*. This is due to the fact that on graphs with large out-degree, such as the pruned graphs, the cost of repeatedly scanning the source vector in *Haveliwala's Algorithm* is only moderate compared to the cost of reading the much larger link data, at least on the limited data sizes we are using, while for sparser graphs it is more significant.

6.5 Results for Topic-Sensitive Pagerank

As described in Section 2, the *Topic-Sensitive Pagerank* scheme involves running multiple, slightly modified Pagerank computations on the web graph, where each computation is biased with respect to a different topic. This bias towards a topic is achieved by a very slight modification of the way in

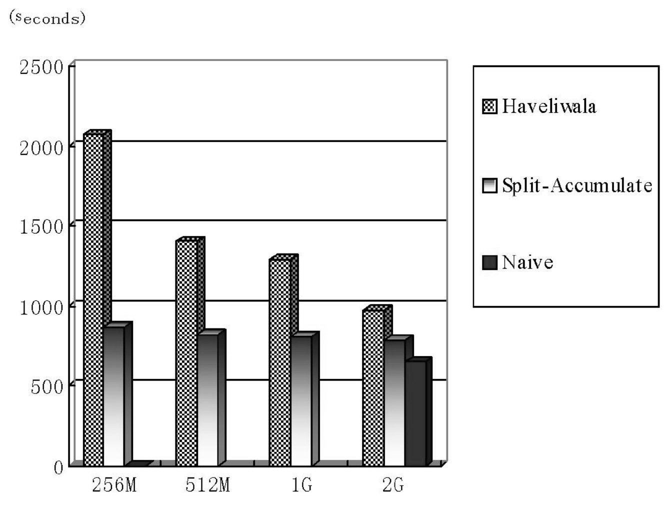


Figure 9: Running times in seconds versus memory size for graph with backlinks.

which the random jump with probability $(1-\alpha)$ is performed, requiring only very minor changes in the implementations.

Thus, if we have k topics in our *Topic-Sensitive Pagerank* scheme, then one approach is to run Pagerank for k times. An alternative approach is to simultaneously perform all k computations, as follows. We simply maintain for each node not one rank value, but a vector of k rank values, one for each topic. Since all Pagerank computations are on the same underlying graph, we can simultaneously update all k values in each iteration. This scheme can be applied to all 4 algorithms that we described; the net effect is that of a Pagerank computation on a graph with k times as many nodes but the same number of edges. Thus, we are now dealing with a scenario where even for larger amounts of memory, we cannot hope to keep the rank vectors for all k topics in main memory.

We experimented with three different schemes for *Topic-Sensitive Pagerank*. (1) *Haveliwala's Algorithm*, modified to simultaneously compute Pagerank on all k topics, (2) the *Split-Accumulate Algorithm*, modified to simultaneously compute Pagerank on all k topics, and (3) a modified version of the *Naive Algorithm*, where we run k/c consecutive computations but where the rank vectors for c topics are computed simultaneously and thus have to fit in main memory. We used the real graph with 45 million nodes and 653 million edges for these experiments.

Figure 10 shows the results for 10 and 20 topics, with the algorithm configured for $256MB$ and $512MB$ of memory. (Note that for these runs we did not modify the physical memory of the machine, similar as in the case of the earlier $32MB$ and $64MB$ runs.) The value c in the modified *Naive Algorithm* was 1 for the case of $256MB$, and 2 for the case of $512MB$ of memory. We note that the *Split-Accumulate Algorithm* is significantly more efficient then *Haveliwala's Algorithm* and the *Naive Algorithm*.

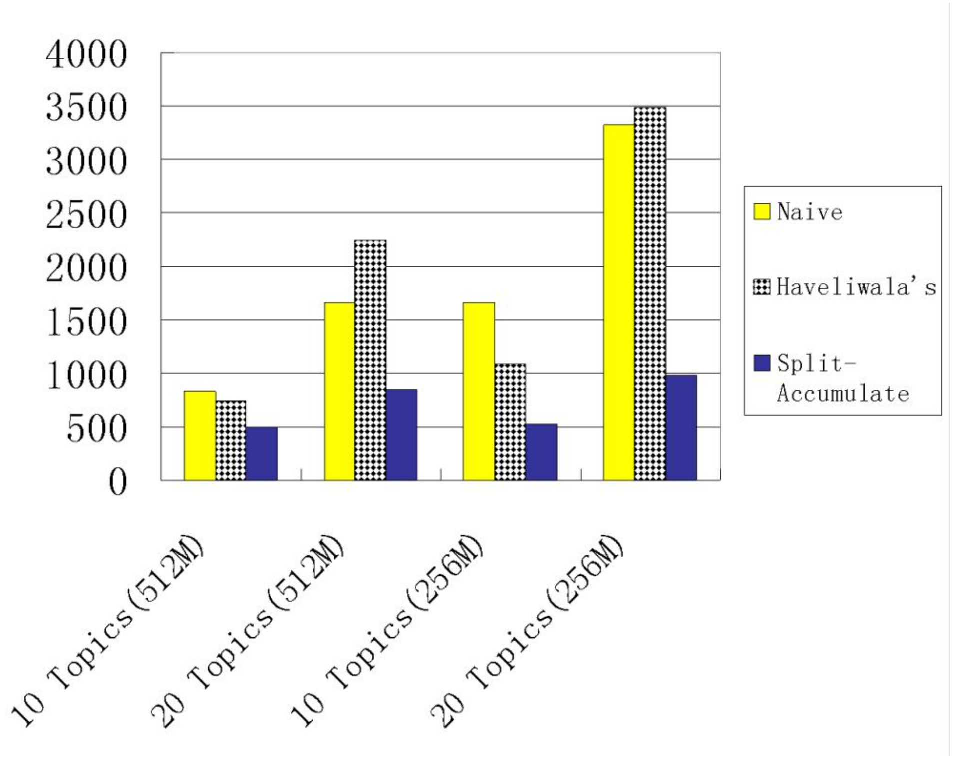


Figure 10: Running times in seconds for Topic-Sensitive Pagerank with 10 and 20 topics on $256MB$ and $512MB$ of memory.

7 Conclusions and Future Work

In this paper, we have derived new I/O-efficient algorithms for Pagerank based on techniques proposed for out-of-core graph algorithms. Our experiments show that there is significant benefit over previous algorithms for Pagerank if the data size is significantly larger than main memory, and also in the case of the recently proposed topic-sensitive version of Pagerank. However, as we admit, under many realistic scenarios the existing algorithms will do just fine given the availability of machines with very large amounts of main memory. Thus, our experimental results show that techniques designed for massive amounts of data are not always a good choice for the more common case where the data size is only slightly larger than memory, e.g., by some small constant factor larger. Most theoretical approaches assume a large gap between data size and memory size; see [1] for an exception.

There are many open challenges in the general area of link-based ranking, and we expect many new schemes to be proposed. We are particularly interested in topic-sensitive schemes such as those in [24, 38] that employ off-line preprocessing in order to allow higher throughput at query time, as opposed to HITS and related schemes that are processed at query time.

Acknowledgements:

This work was supported by NSF CAREER Award NSF CCR-0093400 and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University, and by equipment grants from Sun Microsystems and Intel Corporation. Yen-Yu Chen was also supported by a Sun Foundation Fellowship, and Qingqing Gan was partially supported by NSF Awards CCR-9972568 and ITR-0081964.

References

[1] J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. In *European Symposium on Algorithms*, pages 332–343, 1998.

[2] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Proc. of the IEEE Data Compression Conference (DCC),* March 2001.

[3] A. Arasu, J. Cho, H. Garcia-Molina, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technologies*, 1(1), June 2001.

[4] A. Arasu, J. Novak, Tomkins A, and J. Tomlin. Pagerank computation and the structure of the web: Experiments and algorithms. In *Poster presentation at the 11th Int. World Wide Web Conference,* May 2002.

[5] Ask Jeeves, Inc. Teoma search engine. `http://www.teoma.com`.

[6] B. Awerbuch and T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *IEEE Symp. on Foundations of Computer Science,* pages 459–468, 1993.

[7] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[8] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addision Wesley, 1999.

[9] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: Fast access to linkage information on the web. In *7th Int. World Wide Web Conference,* May 1998.

[10] K. Bharat and M. Henzinger. Improved algorithms for topic distillation in a hyperlinked environment. In *Proc. 21st Int. Conf. on Research and Development in Inf. Retrieval (SIGIR)*, August 1998.

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

[12] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web: experiments and models. In *9th Int. World Wide Web Conference*, 2000.

[13] S. Chakrabarti, B. Dom, D. Gibson, J. Kleinberg, P. Raghavan, and S. Rajagopalan. Automatic resource list compilation by analyzing hyperlink structure and associated text. In *Proc. of the 7th Int. World Wide Web Conference,* May 1998.

[14] S. Chakrabarti, B. Dom, and P. Indyk. Enhanced hypertext categorization using hyperlinks. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 307–318, June 1998.

[15] S. Chakrabarti, B. Dom, R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, David Gibson, and J. Kleinberg. Mining the web's link structure. *IEEE Computer*, 32(8):60–67, 1999.

[16] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proc. of the 8th Int. World Wide Web Conference,* May 1999.

[17] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, and J. Vitter. External-memory graph algorithms. In *Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1995.

[18] S. Chien, C. Dwork, S. Kumar, and D. Sivakumar. Towards exploiting link evolution. Unpublished manuscript, 2001.

[19] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. In *7th Int. World Wide Web Conference*, May 1998.

[20] J. Feigenbaum. Massive graphs: Algorithms, applications, and open problems. Invited Lecture at the 1999 SIAM Annual Meeting.

[21] G. Flake, S. Lawrence, L. Giles, and F. Coetzee. Self-organization and identification of web communities. *IEEE Computer*, pages 66–71, 2002.

[22] J. Guillaume, M. Latapy, and L. Viennot. Efficient and simple encodings for the web graph. 2001. Unpublished manuscript.

[23] T.H. Haveliwala. Efficient computation of pagerank. Technical report, Stanford University, October 1999. Available at `http://dbpubs.stanford.edu:8090/pub/1999-31`.

[24] T.H. Haveliwala. Topic-sensitive pagerank. In *Proc. of the 11th Int. World Wide Web Conference*, May 2002.

[25] International WWW Conference Committee. Proceedings of the World Wide Web Conferences, 1994–2002. `http://www.iw3c2.org/Conferences/`.

[26] A. Kamath, O. Palmon, and S. Plotkin. Fast approximation algorithm for minimum cost multicommodity flow. In *ACM-SIAM Symp. on Discrete Algorithms*, 1995.

[27] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Proc. of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 668–677, January 1998.

[28] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large-scale knowledge bases from the web. In *Proc. of the 25th Int. Conf. on Very Large Data Bases*, September 1999.

[29] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In *Proc. of the 8th Int. World Wide Web Conference*, May 1999.

[30] R. Lempel and S. Moran. The Stochastic Approach for Link-Structure Analysis (SALSA) and the TKC Effect. In *Proc. of the 9th Int. World Wide Web Conference*, May 2000.

[31] M. Marchiori. The quest for correct information on the web: Hyper search engines. In *Proc. of the Sixth Int. World Wide Web Conference*, April 1997.

[32] S. Muthukrishnan and T. Suel. Second-order methods for distributed approximate single- and multicommodity flow. In *2nd Int. Workshop on Randomization and Approximation Techniques in Computer Science*, pages 369–383. Springer LNCS, 1998.

[33] M. Najork and J. Wiener. Breadth-first search crawling yields high-quality pages. In *10th Int. World Wide Web Conference*, 2001.

[34] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Computer Science Department, Stanford University, 1999. Available at `http://dbpubs.stanford.edu:8090/pub/1999-66`.

[35] G. Pandurangan, P. Raghavan, and E. Upfal. Using pagerank to characterize web structure. In *Proc. of the 8th Annual Int. Computing and Combinatorics Conference (COCOON)*, 2002.

[36] P. Pirolli, J. Pitkow, and R. Rao. Silk from a sow's ear: Extracting usable structures from the web. In *Proc. of ACM Conf. on Human Factors in Computing Systems*, April 1996.

[37] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener. The link database: Fast access to graphs of the web. In *Proc. of the IEEE Data Compression Conference (DCC)*, March 2002.

[38] M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in pagerank. In *Advances in Neural Information Processing Systems*, 2002.

[39] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. of the Int. Conf. on Data Engineering*, February 2002.

[40] T. Suel and J. Yuan. Compressing the graph structure of the web. In *Proc. of the IEEE Data Compression Conference (DCC)*, March 2001.

[41] R. Varga. *Matrix Iterative Analysis*. Prentice-Hall, 1962.

[42] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.

[43] D. Zhang and Y. Dong. An efficient algorithm to rank web resources. In *Proc. of the 9th Int. World Wide Web Conference*, May 2000.