

Compressing the Graph Structure of the Web

Torsten Suel *

Jun Yuan *

Abstract

A large amount of research has recently focused on the graph structure (or link structure) of the World Wide Web. This structure has proven to be extremely useful for improving the performance of search engines and other tools for navigating the web. However, since the graphs in these scenarios involve hundreds of millions of nodes and even more edges, highly space-efficient data structures are needed to fit the data in memory. A first step in this direction was done by the DEC Connectivity Server, which stores the graph in compressed form.

In this paper, we describe techniques for compressing the graph structure of the web, and give experimental results of a prototype implementation. We attempt to exploit a variety of different sources of compressibility of these graphs and of the associated set of URLs in order to obtain good compression performance on a large web graph.

1 Introduction

Over the last couple of year, a large number of academic and industrial researchers have studied the problems of searching, exploring, and characterizing the World Wide Web. While many different aspects of the web have been investigated, a significant fraction of recent work has focused on the *graph structure* (or *link structure*) of the Web, i.e., the structure of the graph defined by having a node for each page and a directed edge for each hyperlink between pages.

A number of recent papers have studied this graph, and have looked for ways to exploit its properties to improve the quality of search results. For example, the *Pagerank* algorithm of Brin and Page [4], used in the *Google* search engine, and the *HITS* algorithm of Kleinberg [16] both rank pages according to the number and importance of other pages that link to them. Currently, almost all¹ of the major search engines use information about the link structure in their decision on how to rank search results. Link structure has also been exploited for a variety of other tasks, e.g., finding related pages [3], classifying pages [7], crawling for pages on a particular subject [10, 9], and many other examples. Recent large-scale studies of the web [6, 18, 17] have looked at basic graph-theoretic properties, such as connectivity, diameter, or the existence of bipartite cliques, on subsets of the web consisting of several hundred million nodes and over a billion hyperlinks.

This raises the problem of how to efficiently compute with graphs whose size is significantly larger than the memory of most current workstations. One possibility is to employ I/O-efficient techniques for computing with graphs (e.g., see [12, 19]), but even with these advanced techniques, computation times on large graphs can be prohibitive. The alternative is to use machines with massive amounts of main memory, but this is expensive and not feasible for many smaller research groups. Thus, it would be desirable to construct highly compressed representations of web graphs that can be stored and analyzed in machines with moderate amounts of main memory.

* CIS Department, Polytechnic University, Brooklyn, NY 11201. suel@poly.edu, jyuan@emu.poly.edu

¹See <http://www.searchenginewatch.com/webmasters/features.html>.

A first step in this direction was done with the *Connectivity Server* [2] used by the *AltaVista* search engine, which stores a large subgraph of the web and allows users to query this structure. While the first implementation in [2] uses a fairly inefficient representation of the graph, a newer version is reported in [6] to achieve significant compression compared to a standard adjacency-list representation. No details of the construction were published, but even with these improvements, the study in [6] required almost 10 GB of main memory. Our goal is to find and experiment with techniques that result in improved compression, thus allowing efficient computations with large graphs on more moderately endowed machines.

1.1 Problem Definition

We now define the web graph compression problem. We consider the web graph to consist of a node for each page, and a directed edge for each hyperlink. We assume that each node is labeled with the URL² of the corresponding page, and that the compressed data structure stores this URL in addition to the link structure. We also assume that the outgoing links of a node are an unordered set, and that they can be reordered for improved compression. We would like the compressed structure to support the same operations as an adjacency list representation of a labeled graph, while occupying significantly less space. In particular, we are interested in the following operations:

- (1) **getIndex(url)**: Given a URL, return the index of the corresponding node in the structure.
- (2) **getUrl(index)**: Given the index of a node in the data structure, return its URL.
- (3) **getNeighbors(index)**: Given the index of a node in the data structure, return a list of the indices of all its directed neighbors.

In some applications, it may not be necessary to store the text of the URL together with the graph structure. For example, we could search for certain structures in the graph (e.g., bipartite cliques [18]) using only operation (3) above, which does not require access to the URLs. At the end of the computation, we could then identify the URLs corresponding to the result nodes by using a lookup table on another machine or on disk. In general, we expect (3) to be the most performance-critical operation, while the other two will be used less frequently by most applications. We describe techniques for compressing both link structure and URLs, although our implementation also has the option of building a structure without URLs. The problems of compressing URLs and link structure are largely independent, assuming that we number pages in alphabetically sorted order of the URLs. Such an ordering also tends to maximize compression for both URLs and link structure.

Our current implementation only stores a directed edge from the source to the destination of a hyperlink. There are cases where one would like to also follow links in the reverse direction. We can of course achieve this by adding edges in the reverse direction, using about twice as much space, but we have not yet fully optimized this case. Our current implementation is static and does not allow changes in the graph without rebuilding. We plan to support updates in the future, most likely by periodically merging updates stored in a smaller buffer structure.

1.2 Graph and Machine Sizes

To give the reader a feel for the problem, we now present some numbers concerning the typical data and memory sizes. The current size of the web is estimated at more than 1.2 billion pages and 8 billion hyperlinks. Most major search engines are currently based on a collection of several hundred million web pages³ This is also the size of the graphs used in [6, 18]. In our experiments,

²Uniform Resource Locator, e.g., <http://cis.poly.edu/courses/fall99.htm>.

³The largest search engine, Google, currently (10/2000) contains over 600 million pages, but claims a larger number that includes pages referenced by downloaded pages that have not been downloaded themselves.

we use graph data from a 1998/99 crawl stored at the Internet Archive⁴. This data, which is currently (Fall 2000) the largest collection that is easily accessible to academic researchers, has about 400 million pages and over 2 billion links.

A fairly straightforward implementation of an adjacency list requires about 4 bytes per link (29 bits to address 400 million items, plus overhead), resulting in about 8 GB of link data. An average URL consists of about 50 characters in uncompressed form, though this can be reduced fairly easily to about 13 bytes by exploiting common prefixes in the sorted list of URLs. Thus, this “baseline” implementation would require at least 13 GB for the above crawl represented with edges in only one direction, and 21 GB otherwise.

Our final goal is to be able to handle graphs of this size using machines with at most 2-4 GB of main memory. While this is larger than the main memory of the average workstation, most departments or research groups are probably able to afford a machine in this category. The results reported in this paper achieve about 1.7 bytes per link and 6.5 bytes per URL, for a total of 100 MB on a subgraph with 11 million nodes and 15 million edges. Thus, we would not quite be able to store the above graph in 4GB if we insist on storing the URLs. Future work will hopefully further increase the graph sizes that can be stored.

1.3 Why are Web Graphs Compressible?

Having motivated the need for highly compressed representations of the web graph, we now have to convince ourselves that there is significant potential for achieving compression. Following is an informal description of some of the sources of compressibility in the web graph; a detailed description of the techniques that we use in our implementation is given in Section 3.

As already noted, the size of a URL can be reduced by almost 75% by identifying the length of the common prefix with the previous URL in the sorted order. To compress the remaining text, we can try to identify common substrings, such as `/index`, `/people`, or `.html`. One problem here is how to identify boundaries for suitable substrings⁵. In addition, we could encode single characters or pairs to compress non-frequent text strings. We have to make sure, however, that we can access single nodes in the structure without decoding large chunks of the entire graph.

To compress the link structure, we need to exploit “typical” properties of the web graph. As shown by several studies [17], the web is not just any arbitrary graph, but it has a fairly unique structure. However, this structure does not seem to fit well into any of the families of graphs, such as trees, planar graphs, or graphs of bounded genus or arboricity, that have been studied in the graph compression literature [11, 13, 15]. We can identify a number of possible “sources of compressibility” in the link structure. First, some very popular pages (e.g., `www.yahoo.com`) have much higher in-degree than others, which suggests using appropriate coding for these links. Furthermore, the link structure shows a significant degree of locality: almost three quarters of the links point to pages on the same host, and often to pages that are only a short distance from the source in the sorted order of URLs. Some other sources of compressibility are discussed later.

Given these observations, most people experienced with compression techniques can probably already think of some approaches that one could try. We note that this paper will not present any revolutionary or surprising new compression technique for graphs. Instead, our goal is to design a compression scheme for the web graph based on the careful application and optimization of a variety of known techniques. As we hope to show, the compression of such a graph provides a

⁴<http://www.archive.org>

⁵E.g., should we consider `index.html` as one word or two, and do we include slashes and dots in the words?

number of research challenges, and we hope subsequent work will improve our results.

1.4 Content of this Paper

In this paper, we study the problem of compressing the structure of web graphs, i.e., graphs corresponding to the link structure of the World Wide Web or subsets of it. We assume that both the link structure and the URL strings have to be stored, and that individual nodes and edges can be efficiently accessed in the resulting structure. We describe the techniques that we used in our implementation, and present experimental results on a subset of the web. Note that the results in this version are still very preliminary, and that more final numbers will be available in a Technical Report⁶ during the Spring of 2001.

Section 2 discusses some related work. Section 3 describes our compression scheme and its implementation. Section 4 presents our experimental results. Section 5 describes extensions and optimizations that we are currently working on, and Section 6 offers some concluding remarks.

2 Related Work

We now discuss related work in the data compression and web search areas. For a recent overview of compression techniques, we refer the reader to the textbook by Witten, Moffat, and Bell [20], which contains excellent descriptions of most of the coding techniques that we exploit, including canonical Huffman coding, and techniques for encoding gap sizes. An idea that we apply repeatedly involves the use of *extra bits*, as used, e.g., in `gzip` [14], to reduce the number of codewords needed. As mentioned, there are a number of known techniques for compressing special families of graphs [11, 13, 15]. However, these techniques do not seem to be applicable to web graphs, and we thus rely on standard coding and text compression techniques in our construction.

For recent surveys of information retrieval on the Web with emphasis on link-based methods, we refer to [5, 8]. Examples of ranking techniques based on link structure are the *Pagerank* algorithm of Brin and Page [4] and the HITS algorithm of Kleinberg [16]. Recent large-scale studies of the graph structure of the web are reported in [6, 18, 17].

The *Connectivity Server* of Bharat et al. [2], used in the *AltaVista* search engine, stores a large subgraph of the web in a data structure and provides an interface for remotely querying this structure. The first version of the system, described in [2], required 16 bytes for each URL and 4 bytes for each directed link. The space requirement for the URLs was achieved by stripping common prefixes, as described earlier, and the links were stored as 4-byte pointers. The newer version of the server, as reported in [6], uses 10 bytes for each URL and 3.4 bytes for each bidirectional link, but no implementation details are given. A direct comparison of these numbers with our own results is a bit tricky, as discussed in Section 4, due to differences in the data set. Our work is strongly influenced by the Connectivity Server; in fact, one of our main objectives is to build a similar system that obtains better compression and that is fully accessible to the academic community.

Very recently and independent of our work, Adler and Mitzenmacher [1] have proposed a new technique that exploits the special global structure of the web for compression, and that is based on recent attempts to model the web graph using a new type of random graph model [17]. The idea is to code an adjacency list by referring to the already coded adjacency list of another node that points to many of the same pages. Adler and Mitzenmacher show that combining this with standard Huffman coding results in significant improvements for a graph with global links. The main difference to our work is that [1] focuses on a novel technique that exploits one possible source of

⁶See <http://cis.poly.edu/tr/>.

compressibility in web graphs, and on the algorithmic problems associated with this techniques. In contrast, our goal is to design and engineer a complete tool for web graph compression that is based on a combination of several coding techniques, and that achieves compression due to a variety of properties of the web graph. Integrating the idea in [1] into our system would result in improved compression for global links, and we are currently considering this. One problem is the efficiency of the reference selection of [1] in external memory, since the graphs that we use are significantly larger.

3 Techniques and Implementation

We now describe in detail our compression techniques and their implementation. Throughout the description, we assume that the URLs are indexed from 0 to $n - 1$ in alphabetically sorted order. A overview of our structure is shown in Figure 1(a). We have a Host table that contains the host names in compressed form. The table itself actually consists of concatenated blocks of size 4 MB each. Most of the space in the structure is taken up by the Page table, which also consists of blocks of size 4 MB each. The Page table stores the URL strings, minus the host name prefix, and the adjacency list of each page, both in highly compressed form. The Index table provides pointers into the other tables. More precisely, we have a pointer to the beginning of at least every d -th entry in the Page table, and to the corresponding host. This allows us to search for a particular page, by either index or URL, using a form of binary search.

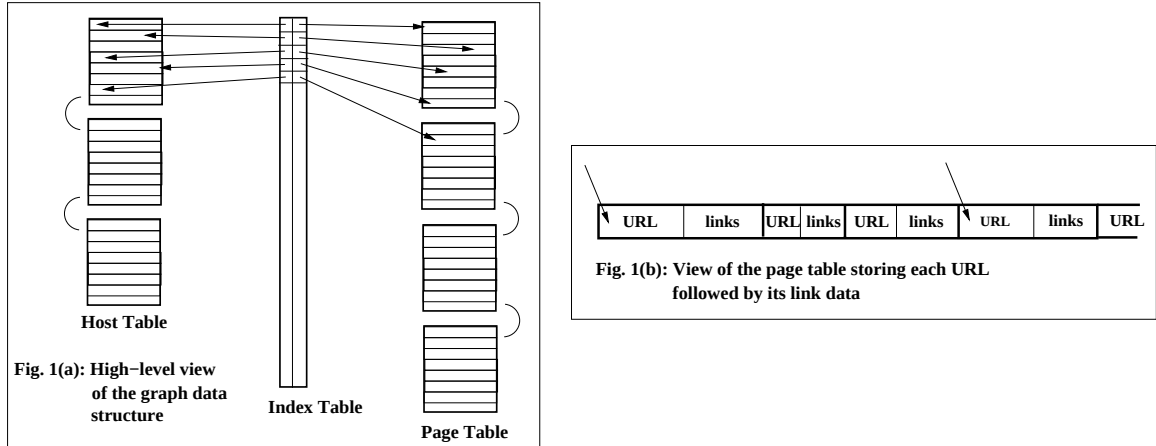


Figure 1(b) shows a more detailed look of the Page table, now shown as a single long array. Pages are stored in alphabetical order by URL, and for each page we store the URL, followed by a list of outgoing links (plus incoming links if specified). Every k th entry has an incoming pointer from the Index table; for these entries, we cannot use the technique of omitting the common prefix with the previous URL, since we want to be able to uncompress these entries without looking at the previous URL. Thus, by changing k we can trade off compression ratio vs. access speed.

3.1 Compressing the URLs and Hosts

As mentioned, we strip the host prefix from each URL, and store it in the Host table. The compression of the Host table is done in basically the same way as the compression of the rest of the URL, described in the following, although it is less critical due to the smaller number of hosts.

We now focus on the compression of the remaining parts of the URL strings, which we will refer to simply as URLs. We assume that any alphabetical ordering is with respect to the original URLs before stripping hosts or any other prefixes, and that each URL is terminated by a special

end symbol @. For each URL i , we now determine the length of the common prefix of i with the previous URL $i - 1$ in the sorted order, denoted by $lcp(i)$, and the change in length of the common prefix, denoted by $dcp(i) = lcp(i) - lcp(i - 1)$. For each URL i , we store $dcp(i)$ as the first field of the page data, using a canonical Huffman code. See the figure below for a typical sequence of URLs before and after stripping common prefixes and replacing them by the values $dcp(i)$.

```
/KAB
/KAB/Contain.html
/KAB/edtrain.html
/Futureshop/music/ramfiles/amanda_int.ram
/deckmanual.html
/users/darmo/TRRC/results/1998/glasscity/gcmres.htm
/users/holben
/users/mharmon
/content/cyberdom/index.html
```

```
0 /KAB
+4 /Contain.html
+1 edtrain.html
-4 Futureshop/music/ramfiles/amanda_int.ram
0 deckmanual.html
0 users/darmo/TRRC/results/1998/glasscity/gcmres.htm
+6 holben
0 mharmon
-6 content/cyberdom/index.html
```

For the remaining strings we now try to identify frequent “words”, where each URL is split into words delimited by slashes and dots⁷. We then determine the w most frequent such words, and encode them with a canonical Huffman code on words. Finally, for any piece of text not yet coded, we use another Huffman code on pairs of 2 consecutive characters. Thus, in summary each URL string is represented by $dcp(i)$ and a sequence of Huffman-encoded words and pairs.

3.2 Compressing the Links

We now describe our compression of the link structure, which is slightly more complicated, and which we focused most of our attention on. We distinguish between two basic kinds of links, *global* links between pages on different hosts, and *local* links between pages on the same host. As in a standard adjacency list, a link will be represented by the index of the link destination.

Global Links: For global links, we first identify the p pages with the highest in-degree, and encode links to these pages using a canonical Huffman code. We call links to such pages *global frequent* links. This results in short encodings for very popular link destinations (e.g., `www.yahoo.com`). For the remaining global links, called *global absolut* links, we use $\log_2(n)$ bits each, where n is the number of pages in the graph. However, if the number of global absolut links from a page is at least 4, then we use an additional encoding based on a Golomb code (see, e.g., [20]). More precisely, let $l_0, \dots, l_{m-1}$ be the list of m global absolut links from the current page, in sorted order by destination. Then we represent each link l_i by the value of $d_i = l_i - l_{i-1}$. We then compute $b = \lceil \log_2(n/m) \rceil$, and encode d_i by a series of $\lfloor d_i/2^b \rfloor$ “1” bits followed by a “0” bit followed by a b -bit representation of $d_i \bmod 2^b$.

Local Links: We again have two classes of links. First, for each host h_i with p_i pages, we determine the $p_i/100$ most popular destinations for local links inside this host. Links to these pages, called *local frequent* links, are encoded using a Huffman code. (The number 100 in the denominator appears to be a good choice, as it balances compression with table space overhead.)

We then look at all the remaining links, called *local distance* links, and collect statistics about the typical distance between source and destination page in the sorted order. As it turns out, a significant number of links go only over a fairly small distance⁸. More precisely, we partition the set of hosts into a number of different classes (about 10 to 15), according to their size, and collect the distance statistics separately for each host class. We then represent each local distance link by coding the distance between source and destination. In a further optimization, not yet reflected in the reported results, we used the gaps between the destinations instead of the distance. In order to limit the number of codewords, and thus the size of the tables, we do not reserve

⁷For example, `faculty/index.html@` consists of the words `faculty`, `/index` and `.html@`.

⁸However, the distribution also has a very significant tail that limits compression.

a codeword for each distance, but instead group distances into intervals of increasing size, and assign an appropriate number of *extra bits* to each group.

Storing the links: We also need to store the number of links and the type of each link in a node. After a few clumsy and inefficient attempts, where we used additional leading bits in each link to describe its type, and an additional bitfield for the total number of outgoing links, we arrived at the following approach. All links in a node are reordered so that we first have all global frequent links, followed by global absolute links, followed by local frequent links, followed by local distance links. We then store the tuple (gf, ga, lf, ld) , containing the number of global frequent, global absolute, local frequent, and local distant links, respectively, after the URL and before the links. The tuples (gf, ga, lf, ld) are encoded using a single Huffman code on the set of such tuples. To limit the number of codewords, we again group classes of adjacent tuples into intervals, in this case rectangles in 4 dimensions, with the size of the intervals increasing with distance from the origin. An appropriate number of *extra bits* is used for intervals containing more than one tuple.

3.3 Implementation

Preprocessing: The data from the Internet Archive was initially stored in a meta data format used by Alexa Inc. for its crawls. In this format, links are identified by destination URL, and nodes are listed in arbitrary order. Using several I/O-efficient sorting and merging steps, we transformed this data into a format where the URLs are stored in alphabetical order in one file, already compressed by stripping common prefixes, and the links are stored sorted by source in another file, where every link is represented as a (source,destination) pair of line numbers in the URL file. This is the format that we assume for the input of our compression tool. The preprocessing takes significantly more time than the actual compression phase, due to the large initial data size.

Gathering statistics: We first performed several scans over the URL and link data to gather statistics for our coding. In a first pass over the URL file, word frequencies are counted and used to construct the Huffman code for frequent words in the host and URL string. In a second pass, the statistics for the pairwise encoding are collected on parts of the host and URL strings not covered by the frequent words. In a first pass over the link file, we gather statistics for the most popular hyperlink destinations, both global and inside each host. Using this data, we can decide after this pass which of the four categories (global frequent, global absolute, local frequent, local distance) each link belongs to. In the second pass, we collect the distance statistics for the local distance links, and the statistics for the tuple frequencies for (gf, ga, lf, ld) .

Building the compressed structure: We then build the structure in a single coordinated pass over both URL and link data. The entire data structure in Figure 1 is built from top to bottom during this pass; this allows us to build structures larger than main memory and write them out to disk. As parameters, we chose 100000 frequent words and 100000 global frequent link destinations. For the local distance links, we considered 12 different host size classes, and used several hundred codewords for each class. We used 10000 codewords to model the tuple statistics (10 intervals in each dimension). All components were implemented in C resulting in about 5000 lines of code.

4 Experimental Results

We now provide an overview of our experimental results on compression ratio and access speed. We used data from a 1998/99 crawl of about 400 million pages stored at the Internet Archive. However, for the experiments reported in this version, we did not run our code on the full set of pages, but instead used a subset consisting of 11 million pages and about 15 million links. We also prepared to perform runs on a larger set consisting of 125 million pages and about 190 million

links. Unfortunately, we were unable to access the Internet Archive for the last several weeks, and thus only very preliminary results on the smaller set of 11 million URLs are reported. As mentioned before, more final numbers will be available in a Technical Report⁹ during the Spring of 2001.

A few comments are in order about the scaling of such data sets, and the ratio of pages to links. Our data sets contain nodes for actually crawled pages as well as pages pointed to by crawled pages. This leads to a larger pages-to-links ratio, compared to the web as a whole where we have about 7 to 8 links for every page. The ratio also depends on the crawling strategy and the size of the data set, since larger sets tend to have a smaller pages-to-links ratio. Most studies of the web perform a preprocessing step that prunes some of the nodes; e.g., [6] keeps only pages that were either crawled, or referenced at least 5 times by other pages. We have not yet incorporated such a cleaning phase, and thus it is difficult to compare our numbers directly to those reported in [6] for the new version of the *Connectivity Server*.

4.1 Compression Ratio

We now present our results on the compression ratio, shown in Figure 3(a). As mentioned, the results are very preliminary at the moment. The baseline implementation already saves space by omitting common prefixes between subsequent URLs, and uses 24 bits to address the 11 million nodes. (This is somewhat optimistic since we need extra bits to mark the end of an adjacency list.) Our method compresses each URL to about 6.49 bytes and each edge to about 1.74 bytes on average. The compression ratio is not very impressive when compared, e.g., to results in text or image compression, but this is not surprising given the data set. Figure 3(b) shows the number of links in each of the 4 classes, and the compression for each class. Each of our techniques contributes to the compression, but no technique alone gives good compression on the entire graph.

	Nodes	Edges	Total Size
Number	11556269	14709649	
Baseline	13 bytes	3 bytes	194.4 MB
Compressed	6.49 bytes	1.74 bytes	100.6 MB
Ratio	49.9 %	58 %	51.7 %

Type of Link	Number	Size
Global Absolute	1636836	2.64
Global Frequent	2308253	1.97
Local Frequent	1015728	1.06
Local Distance	9748832	1.30

Figure 3(a): Compression results on small graph

Figure 3(b): Results for different types of links

Comparing these numbers to those claimed for the new version of the Connectivity Server in [6] is difficult, since no details were published, and since the graphs are quite different. The numbers in [6] are 10 bytes per URL and 3.4 bytes for a link in both directions. Without implementation details, it is difficult to predict how the bidirectional links affect compression. The data sets in [6] are larger, and thus more bits would be needed for global absolute links. On the other hand, the data in [6] has significantly more links, which is likely to reduce the cost per link. For the URLs, more pages would likely result in better compression. In summary, we feel that we cannot make a valid comparison yet.

4.2 Access Speed

We now give performance numbers for the three data access operations defined in Subsection 1.1. These numbers are also preliminary since we have not optimized for access times yet. We looked at two values for k , the number of pages between two pointers from the Index table into the Page

⁹Check <http://cis.poly.edu/tr/> or contact the authors.

table. The times for *getNeighbor* are per link, but the time really consists of a fixed cost for finding the node using binary search, plus a lower cost for decoding each link in the node. We plan to measure this effect more precisely in the final version.

	getUrl	getIndex	getNeighbors
k = 20	0.26 ms	1.7 ms	0.26 ms/link
k = 40	0.4 ms	1.7ms	0.57 ms/link

Figure 4: Time for access to the data

5 Current and Future Extensions

We now list a few additional ideas that we are currently implementing or considering for implementation.

- (1) **Locality:** we currently have only one level of locality, within a host, which probably captures most of the potential benefits. However, one could consider additional levels, e.g., internet domains or directories inside a host, to improve compression of links within these levels.
- (2) **Representation of Huffman tables:** since we are using quite a number of Huffman codes, it would make sense to better optimize the representation.
- (3) **Exploiting URL structure for link compression:** one might try to use the structure of the URLs to improve link compression; however, this requires that the URLs are stored as well.
- (4) **Integrating the idea of Adler and Mitzenmacher [1]:** this would result in improved compression for global links, but might significantly increase the time for building the structure.
- (5) **Correlations among local links:** We are currently using Huffman coding for frequent local links. However, there are often strong correlations between these links, e.g., when many pages of a site have the same menu of navigational links. One could apply the technique in [1] here, but we believe that explicitly recognizing such sets might be better for local links.
- (6) **Bidirectional links:** An improvement could be achieved here by only storing the forward-going direction for very "short" links that run between two pointers into the page table, since the other direction can be found while traversing the structure.
- (7) **Updates:** we plan to allow insertions and deletions of nodes and edges, most likely by storing updates in a smaller buffer structure that is periodically merged into the main structure.

6 Conclusions

In this paper, we have described techniques for compressing the link structure of the web, and have presented preliminary experimental results on compression ratio and access speed. Beyond the extensions listed in the previous section, which we are currently working on, there are other more fundamental open questions in the context of efficient computing with large web graphs.

An obvious question is whether there are other more novel techniques that offer substantial improvements in compression, beyond those achievable by optimizing the types of coding schemes we use in our work. The work in [1] provides one such technique for global links. We are particularly interested in a better coding of local links. Most of the recent studies of the web graph have focused on the global link structure, often even removing local links in a preprocessing phase.

It would be interesting to study the representation of simplified versions of the web graph, e.g., graphs where certain clusters of nodes are collapsed into a single node. Such representations could

be significantly more concise, while still supporting many interesting applications. Investigating persistent graph structures that retain a history of updates might also be of interest. Finally, our main interest is in using our implementation to study the web and its evolution.

Acknowledgements: We acknowledge the Internet Archive (<http://www.archive.org>) for providing access to the web graph data. We also thank Nasir Memon for helpful discussions.

References

- [1] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Proc. of the IEEE Data Compression Conference (DCC)*, March 2001. To appear.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] A. Broder and M. R. Henzinger. Information retrieval on the web. In *Proc. of the 30th IEEE Symp. on Foundations of Computer Science*, October 1998.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] S. Chakrabarti, M. van den Berg, and B. Dom. Distributed hypertext resource discovery through examples. In *Proc. of 25th Int. Conf. on Very Large Data Bases*, pages 375–386, September 1999.
- [10] 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 (WWW8)*, May 1999.
- [11] S. Chen and J. Reif. Efficient lossless compression of trees and graphs. In *IEEE Data Compression Conference (DCC)*, 1996.
- [12] 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.
- [13] N. Deo and B. Litow. A structural approach to graph compression. In *Proc. of the MFCS Workshop on Communications*, pages 91–101, 1998.
- [14] J. Gailly. gzip compression utility. Available at <http://www.gzip.org>.
- [15] K. Keeler and J. Westbrook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58:239–252, 1995.
- [16] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Proc. of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 668–677, January 1998.
- [17] 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.
- [18] 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 (WWW8)*, May 1999.
- [19] J. Vitter. External memory algorithms and data structures. In *External Memory Algorithms and Visualization, DIMACS Series on Discrete Mathematics and Theoretical Computer Science*. AMS, 1999i.
- [20] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.