The WebGraph Framework I: Compression Techniques*

Paolo Boldi
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39/41, I-20135 Milano, Italy
boldi@dsi.unimi.it

Sebastiano Vigna
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39/41, I-20135 Milano, Italy
vigna@acm.org

ABSTRACT

Studying Web graphs is often difficult due to their large size. Recently, several proposals have been published about various techniques that allow to store a Web graph in memory in a limited space, exploiting the inner redundancies of the Web. The WebGraph framework is a suite of codes, algorithms and tools that aims at making it easy to manipulate large Web graphs. This papers presents the compression techniques used in WebGraph, which are centred around referentiation and intervalisation (which in turn are dual to each other). WebGraph can compress the WebBase graph (118 Mnodes, 1 Glinks) in as little as 3.08 bits per link, and its transposed version in as little as 2.89 bits per link.

Categories and Subject Descriptors

E.2 [**Data**]: Data Storage Representations; E.4 [**Data**]: Coding and Information Theory; H.3 [**Information Systems**]: Information Storage and Retrieval

General Terms

Algorithms, Experimentation, Measurement

Keywords

Web graph, compression

1. INTRODUCTION

In the last few years, the World Wide Web has become the focus of an intense research activity, performed by both academic and industrial research centres; this activity is mainly aimed at developing efficient techniques to retrieve information over the Web, using some form of exploration or search that is especially tailored to the specific hypertextual structure of the Web itself. These techniques find many potential and actual applications, for example, in search engines, in the design of effective crawlers, in determining cybercommunities, etc.

In many cases, most of the information one needs to perform a search is contained in the structure of the *Web graph* (or link graph), that is the graph having a node for each URL, and a (directed) arc from node x to node y whenever there is a hyperlink in page x leading to page y.

*This work has been partially supported by a "Finanziamento per grandi e mega attrezzature scientifiche" of the Università degli Studi di Milano.

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

Needless to say, the Web graph is a huge object to deal with: it currently contains some 3 billion nodes, and more than 50 billion arcs (and these estimates are just lower bounds, as they are obtained from search engines, which index just a part of the Web).

In this paper, we present new compression techniques that are used in WebGraph to represent compactly Web graphs. WebGraph is a framework that provides simple methods to manage very large graphs. More precisely, it is currently made of:

1. A set of flat codes, called ζ *codes*, which are particularly suitable for storing Web graphs (or, in general, integers with a power law distribution in a certain exponent range). The fact that these codes work well can be easily tested empirically; a more detailed mathematical analysis can be found in the companion paper [6].

2. Algorithms for compressing Web graphs that exploit gap compression (as in the Connectivity Server [2]), referentiation (à la LINK [11]), intervalisation and ζ codes to provide a high compression ratio, and algorithms for accessing a compressed graph without actually decompressing it, using lazy techniques that delay the decompression until it is actually necessary. The algorithmic part of WebGraph is the topic of this paper.

3. A complete, documented implementation of the algorithms above in Java, contained in the package `it.unimi.dsi.webgraph`. Besides a clearly defined API, the package contains classes that allow one to modify (e.g., transpose) or recompress a graph, so to experiment with various settings.

4. Data sets for very large graphs (e.g., a billion of links). These data were either gathered from public sources (such as WebBase [7]) or obtained with UbiCrawler [5, 4].

One of the features of the WebGraph compression format is that it is devised to compress efficiently not only the Web graph, but also its transposed graph (i.e., a graph with the same nodes, but with the direction of all arcs reversed). A compact representation of the transposed graph is essential in the study of several advanced ranking algorithm (e.g., HITS [8]): the literature often reports that the transposed graph is more "entropic", and thus more difficult to compress than the graph itself [10, 11], but we shall see that in the WebGraph framework transposed graphs actually compress *better*.

2. THE WEB GRAPH

The *Web graph* relative to a certain set of URLs is a directed graph having those URLs as nodes, and with an arc from x to y whenever page x contains a hyperlink toward page y. When trying

to devise a compression mechanism to store a Web graph efficiently we can exploit some empirical observations about the structure of hyperlinks in a typical subset of the Web.

The features of the links of a Web graph that are usually quoted are *locality* and *similarity*, which were originally exploited by the Connectivity Server [2] and by the LINK database [11].

1. **Locality**. Most links contained in a page have a *navigational* nature: they lead the user to some other pages within the same host ("home", "next", "previous", "up" etc.); if we compare the source and target URLs of these links, we observe that they share a long common prefix; said otherwise, if URLs are sorted *lexicographically*, the index of source and target are close to each other.

2. **Similarity**. Pages that occur close to each other (in lexicographic order) tend to have many common successors; this is because many navigational links are the same within the same local cluster of pages, and even non-navigational links are often copied from one page to another within the same host.

These features suggest to use techniques borrowed from full-text indexing for storing increasing sequences of integers with small gaps, and moreover inspired the *reference compression* techniques discussed in [11, 1]. Since several successor lists are similar, one can specify the successor list of a node by copying part of a previous list, and adding whatever remains. This is achieved using a list of bits, one for each successor in the referenced list, which tell whether the successor should be copied or not, or using other techniques (such as explicit deletion lists [11]).

The empirical analysis at the base of WebGraph's compression techniques evidenced two additional facts:

1. **Similarity is much more concentrated than it was previously thought**. Either two lists have almost nothing in common, or they share large segments of their successor lists. This implies that the one-bit-per-link scheme used in reference compression may be refined to a *copy-block list* scheme, in which the links to be copied are specified by means of interval lengths (this corresponds essentially to a run-length encoding of the reference bits).

2. **Consecutivity is common**. It can be observed that many links within a page are consecutive (with respect to the lexicographic order); this is due to two distinct phenomena. First of all, most pages contain sets of navigational links which point to a fixed level of the hierarchy. Since the hierarchical nature of a site is usually reflected in the hierarchical nature of URLs, links in pages at the bottom of the hierarchy tend to be adjacent in lexicographic order. Second, in the transposed Web graph pages that are high in the site hierarchy (e.g., the home page) are pointed to by most pages of the site. This, of course, gives also rise to large intervals.

3. **Consecutivity is the dual of distance-one similarity**. If a graph is easily compressible using similarity at distance one (i.e., exploiting similarity with the successor list of the previous node in lexicographical ordering), its transpose must sport large intervals of consecutive links, and viceversa, as a node that is common among two or more consecutive successor lists at distance one is reflected by a corresponding interval of length two or more in the transposed graph.

As an example, see Figure 1 and Figure 2, which show the distribution of gaps in increasing sequences of successors for a snapshot of the `.uk` domain: gaps are regularly distributed along a power-law distribution (a fact which is exploited in [6]), but the gap 1 lies over the interpolating line (i.e., intervals are very frequent; this phenomenon is particularly evident in the transposed graph).

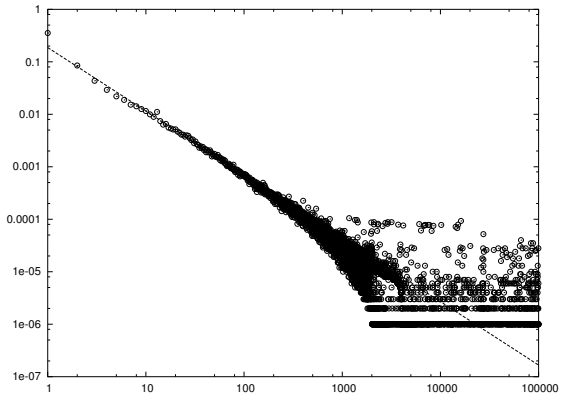

Figure 1: Distribution of gaps in a 18.5 Mpages snapshot of the `.uk` domain. The scale is logarithmic on both axes, and the line displays a power law with exponent 1.21.

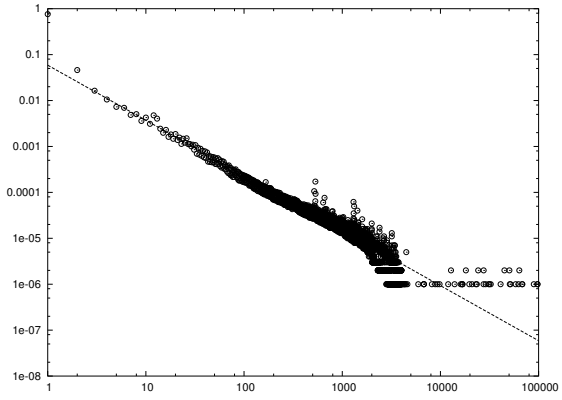

Figure 2: Distribution of gaps in the transpose of a 18.5 Mpages snapshot of the `.uk` domain. The scale is logarithmic on both axes, and the line displays a power law with exponent 1.20 (modulo a scaling factor).

The considerations above suggest that a compression format for the Web graph and its transpose *should take into consideration at the same time similarity and consecutivity*. As we shall see, our compression format takes indeed these phenomena into account, obtaining a high compression ratio.

3. THE COMPRESSION FORMAT

Throughout this section, whenever we say that an integer is part of the compression format, we mean that a suitable instantaneous coding must be chosen for the integer (WebGraph allows to choose among several codes). For consistency, we assume that *all codings encode natural numbers*, that is, the first code is for 0, the second one for 1 and so on. This is natural for some codings (e.g., Golomb) and less natural for other codings (e.g., γ), which must be shifted suitably. However, this allows to treat uniformly different coding techniques. We remark that this convention is the one adopted in

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

Table 1: Naive representation using outdegrees and adjacency lists.

Node	Outdegree	Successors
...	...	...
15	11	3, 1, 0, 0, 0, 0, 3, 0, 178, 111, 718
16	10	1, 0, 0, 4, 0, 0, 290, 0, 0, 2723
17	0	
18	5	9, 1, 0, 0, 32
...	...	...

Table 2: Representation using gaps.

MG4J[1], but it is different from the one used in the companion paper [6].

Naive representation. Suppose that we are interested in representing a Web graph relative to some set of N URLs; the graph nodes will be numbered from 0 to $N-1$ according to the *lexicographic* ordering of URLs. We let $S(x)$ denote the set of successors of node x (i.e., the set of all nodes y such that there is an arc from x to y)[2].

We wish to represent the graph using adjacency lists: in other words, the graph will be coded as the sequence of adjacency lists of nodes 0, 1, etc., each preceded with the outdegree of the corresponding node, to make it self-delimiting. This naive representation is exemplified in Table 1.

Using gaps. The example shows the locality phenomenon discussed above; locality suggests that we should represent each list of successors as a list of gaps (as pioneered by the Connectivity Server). More precisely, if $S(x) = (s_1, \ldots, s_k)$, we will represent it as $(s_1 - x, s_2 - s_1 - 1, s_3 - s_2 - 1, \ldots, s_k - s_{k-1} - 1)$ instead; note that all the integers obtained in this way are non-negative, except possibly for the first one. Since we do not want to deal with negative numbers, we will code the first element suitably, using the map $\nu : \boldsymbol{Z} \to \boldsymbol{N}$

$$\nu(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ 2|x| - 1 & \text{if } x < 0. \end{cases}$$

In Table 2 you can see this modified representation using gaps.

Reference compression. Another possible way to improve the compression ratio is to exploit similarity: instead of representing the adjacency list $S(x)$ directly, we can code it as a "modified" version of some previous list $S(y)$, called the *reference list*. The difference $x - y$ is called the *reference number*. As we already mentioned, this results usually in *reference compression*, in which a sequence of bits, one of each successor in the reference list, tells whether the corresponding successor of y is also a successor of x.

More precisely, the representation of $S(x)$ with respect to $S(y)$ is made of two parts: a sequence of $|S(y)|$ bits, called the *copy list*, and the list of integers $S(x) \setminus S(y)$, called the *list of extra nodes*. The copy list specifies which of the links contained in the reference list should be copied: it will contain 1 at the i-th position iff the i-th entry of list $S(y)$ also appears in $S(x)$.

The resulting representation is shown in Table 3; note that every copy list is preceded by a the reference number: if the reference is r for list $S(x)$, it means that the compression is relative to list $S(x-r)$ (if $r = 0$ then we are not compressing the list by reference).

The choice of r is critical, here; we assume that there is a fixed parameter $W > 0$ (called the *window size*), and r is chosen as the value between 0 and W that gives the best compression. A large value of W is likely to produce better compression ratios (simply because it enlarges the set of possible reference lists); the price for this improvement is a slower and more memory-consuming compression and decompression.

Several forms of reference compression are used in different versions of the LINK database; moreover, reference compression is analysed from a theoretical viewpoint in [1].

Differential compression. WebGraph introduces *differential compression*, in which the differences with $S(y)$ are recorded by a sequence of copy blocks: we look at the copy list as an alternating sequence of 1- and 0-blocks, and specify the length of each block (decremented by one, for all blocks except the first one). This sequence of integers is preceded by a *block count* telling the number of blocks that will follow. We will always omit the last block from the list of blocks, because its value can be deduced from the block count and from the outdegree of the reference node. The resulting scheme is exemplified in Table 4.

Note that the first copy block always refers to a 1-block (so, the first copy block is 0 if the copy list starts with a 0). Using typical codes, such as γ coding, copying entirely a list costs one bit: more generally, differential compression allows to code a link in *less than one bit*, something which is impossible with plain reference compression.

Using intervals to exploit consecutivity. As we observed in Section 2, consecutivity is frequent among extra nodes; hence, instead of compressing them directly using the gap technique, we first isolate the subsequences corresponding to integer intervals. Only intervals whose length[1] is not below a certain threshold $L_{\min}$ are considered.

Hence, each list of extra nodes will be compressed as follows:

- a list of integer intervals; each interval is represented by its left extreme and by its length; left extremes are compressed using the differences between each left extreme and the previous right extreme minus 2 (because there must be at least one integer between the end of an interval and the beginning of the next one); interval lengths are decremented by the threshold $L_{\min}$.
- a list of *residuals* (the remaining integers), compressed using differences.

Table 5 shows the resulting representation assuming that the interval threshold is 2. For example, consider the list of remaining

[1]MG4J (Managing Gigabytes for Java) is a package providing bit-level I/O; it can be downloaded at `http://mg4j.dsi.unimi.it/`.

[2]In this paper, with some abuse of notation, we will not distinguish between a set of integers and the corresponding list of integers in increasing order. Hence, if $A = \{45, 12, 378, 40\}$ we will also use the notation A for the list $(12, 40, 45, 378)$. Note, however, that some algorithms on the Web graph require that the original order of the links be preserved. In this case, techniques described in this paper are not viable.

[1]The length of an integer interval is the number of integers it contains.

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Table 3: Representation using copy lists.

Node	Outd.	Ref.	# blocks	Copy blocks	Extra nodes
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	7	0, 0, 2, 1, 1, 0, 0	22, 316, 317, 3041
17	0				
18	5	3	1	4	50
...	...	...	...	...	...

Table 4: Representation using copy blocks.

nodes for node 15:

- there are two intervals ([15, 19] and [23, 24]); the left extremes, compressed as explained above, give $15 - 15 = 0$ (the first extreme is represented as a difference with respect to the node itself) and $23-19-2 = 2$ (the other extremes are represented as difference with respect to the previous right extreme minus 2); the lengths are 5 and 2, but should be reduced by the threshold (2), which gives 3 and 0 respectively;
- the residuals are 13, 203, 315, 1034, which gives $13 - 15 = -2$ (encoded as $2 \cdot |-2| + 1 = 5$), $203 - 13 - 1 = 189$, $315 - 203 - 1 = 111$ and $1034 - 315 = 718$.

The final format. The WebGraph format can be summarised as follows:

$$d\left[\overbrace{r\big[b\ B_1 \cdots B_b\ \big]_{r>0}}^{W>0}\ \left[\overbrace{i\ E_1 L_1 \cdots E_i L_i}^{L_{\min}<\infty}\ R_1\ \cdots\ R_k\right]_{\beta<d}\right]_{d>0}$$

where the data are described in Table 6. We denote with a subscript to brackets the condition on previous data for a certain part of the coding to be present, and with an overbrace the analogous conditions on compression parameters.

The value β represents the number of successors that have been copied from the reference list, and may be computed as follows:

$$\beta = \begin{cases} 0 & \text{if } r = 0 \\ \sum_{1\le h\le b, h \text{ odd}} B_h & \text{if } r = 0 \text{ and } b \text{ odd} \\ |S(x-r)| - \sum_{1\le h\le b, h \text{ even}} B_h & \text{if } r = 0 \text{ and } b \text{ even.} \end{cases}$$

The only datum that is always present is the outdegree d: if $d = 0$, no other data needs to be stored for that adjacency list. The part relative to differential compression is, of course, omitted if $W = 0$, because in that case differential compression is inhibited; moreover, if $r = 0$ (i.e., if we have decided not to compress the list differentially), the block part is not stored. The extra part (which stores intervals and residuals) is omitted if $\beta = d$, because in that case all successors have been copied from the reference list. Finally, the part relative to intervals (interval count, left extremes and interval lengths) is omitted if $L_{\min} = \infty$ (which means that we do not want to use interval compression).

Note that the adjacency list is self-delimiting (as long as the outdegree of the reference node is known), because all the integer lists it comprises are preceded by their length, except for the list of residuals whose length can be computed as $k = d - \beta - \sum_{h=0}^{i} L_h$.

4. REFERENCE CHAINS

As we have seen, the overall compression process is based on two parameters, specifying the window size W and the minimum interval length $L_{\min}$. The choice of W determines a trade-off between the compression ratio and the time needed to compress/decompress the graph. A remark is in order, here: say that node x *(directly) refers* to node y iff x has reference $r > 0$ and $x - r = y$; when x refers to y, if we want to access the adjacency list of node x we will first have to decompress the adjacency list of node y.

Of course, $y \geq x - W$, so if we are accessing the adjacency lists in a sequential fashion, we just need to keep a sliding window of the last W adjacency lists. Conversely, if we need a direct random access to adjacency lists, we have to consider carefully the problem of accessing in turn all the lists they (directly or indirectly) refer to. In other words, if we want to access to $S(x)$ and if x directly refers to y, we will first have to access $S(y)$, and if y directly refers to z, we will first have to access $S(z)$ and so on. The sequence of nodes built in this way is called the *reference chain of node* x.

If we put no limit on this, in the worst case accessing to the adjacency list of node x may require a decompression of all lists up to x. This will cause a severe slowdown in the time required to perform random accesses.

Of course, this is not a problem if we only plan to access adjacency lists sequentially (as is the case, for example, in the computation of PageRank [9]); yet, if random access is our primary business, we need to put a limit on the lengths of reference chains.

Thus, there is a further parameter R to our compression algorithm, called the *maximum reference count*. When performing differential compression of $S(x)$, we do not consider all nodes $x - 1, \ldots, x - W$, but only the ones that do not produce reference chains longer than R. A small value for R is likely to produce worst compression, but shorter (random) access times.

The authors of the LINK database report testing on the number of bits per link reached with different values for W and R: here, we provide statistics about the number of bits per link and the reference

Node	Outd.	Ref.	# blocks	Copy blocks	# intervals	Left extremes	Length	Residuals
...	...	...	...	...	...	...	...	...
15	11	0			2	0, 2	3, 0	5, 189, 111, 718
16	10	1	7	0, 0, 2, 1, 1, 0, 0	1	600	0	12, 3018
17	0							
18	5	3	1	4	0			50
...	...	...	...	...	...	...	...	...

Table 5: Representation using intervals (interval threshold is 2).

Datum	Meaning	Notes	Represented as...
d	Outdegree	$d \geq 0$	
r	Reference number	$0 \leq r \leq W$	
b	Block count	$b \geq 0$	
$B_1, \ldots, B_b$	Blocks	$B_1 \geq 0, B_2, \ldots, B_b > 0$	$B_1, B_2 - 1 \ldots, B_b - 1$
i	Interval count	$i \geq 0$	
$E_1, \ldots, E_i$	Left extremes	$E_{k+1} \geq E_k + L_k + 1$	$\nu(E_1 - x), E_2 - E_1 - L_1 - 1, \ldots, E_i - E_{i-1} - L_{i-1} - 1$
$L_1, \ldots, L_i$	Interval lengths	$L_1, \ldots, L_i \geq L_{\min}$	$L_1 - L_{\min}, \ldots, L_i - L_{\min}$
$R_1, \ldots, R_k$	Residuals	$0 \leq R_1 < R_2 < \cdots < R_k$	$\nu(R_1 - x), R_2 - R_1 - 1, \ldots, R_k - R_{k-1} - 1$

Table 6: Data describing the adjacency list of node x.

chains. The results are summarised in Table 7: one can easily see that $R = \infty$ gives excellent results, although even when $R = 1$ we improve significantly over existing techniques[2].

For comparison, version 3 of the LINK database provides a compression of 5.61 bits per link for a Web graph of 61 Mnodes and 1 Glinks graph, and 5.66 bits per link for its transpose, whereas [10] reports 5.07 bits per link and 5.63 bits per link, respectively, for the WebBase graph and its transpose[3]. The figures reported in [12] on a 115 Mpages/1.47 Glinks snapshot taken from the Internet Archive, on the other hand, are much worse (13.92 bits per link; no data is provided for the transpose).

5. THE OFFSET ARRAY

As we have seen, the graph is described as a sequence of adjacency lists; each list is in turn represented by a sequence of natural numbers and, at the very last level, we encode each natural number in these sequences using some kind of self-delimiting bit-encoding (the choice of this encoding for residuals is discussed thoroughly in [6]).

If we want to access the graph in a random fashion, we must keep an auxiliary vector of offsets, that has N entries (one for each node); the entry of index h represents the position (in memory) where the successor list of node h starts. Offsets can be expressed as bit- or byte-displacements, the second option being actually applicable only if each adjacency list is byte-aligned.

The advantage of byte displacement is that a 32-bit offset is sufficient to index 4 GiB of RAM; thus, it can be considered sufficient to exploit the current core memories. Using 32-bit offsets for bit displacement is not sensible, as it would allow indexing just 512 MiB of RAM, which, at the current compression rates, would allow to represent just 150 millions of nodes.

On the other hand, byte displacement requires aligning each successor list, and this requirement is very expensive (more than 10% increase in size); moreover, core memory exceeding 4 GiB is already available with the current technology, and in any case, using 32-bit byte offset displacements does not allow to store more than about 4 billions of nodes: considering the current size of the Web, this solution suffers from an inherent lack of scalability.

These reasons pushed the authors to implement WebGraph using bit displacements. This, of course, requires some considerations about offset representation in memory, as consuming 64 bits for each offset would make the offset array larger than the graph itself. The authors of the LINK database, for instance, propose a complex scheme in which nodes of lower degree get smaller offsets; other proposals require a main part that is kept in main memory, and a part that is kept offline [10].

Our solution rests on a different viewpoint: much in the same way as WebGraph allows to trade off between speed and compression ratio when *compressing* a graph, it also allows to tune speed and offset array size while *decompressing* it. Essentially, if we need to perform random accesses, but we have only a limited amount of central memory, we may load the offset array only partially; more precisely, we only keep the offsets of nodes 0, J, $2J$, ..., for a suitable parameter J (called the *jump*). When we need to access the adjacency list $S(x)$, we use the offset relative to node $J\lfloor x/J \rfloor$, and then we skip over (i.e., read and ignore) x mod J entries sequentially. If J is large, we spare memory but lose time when accessing a node; if J is small, we have faster access times but higher memory consumption. Note that J is not a compression parameter: rather, it is fixed when *reading* the graph into memory.

A minor point to take into consideration here is the following: as explained in the previous sections, each adjacency list is stored in an *almost* self-delimiting format, except for the list of residuals, whose length k depends on the number β of links copied from the reference list, which may in turn depend on the outdegree $|S(x-r)|$ of the reference list itself. Unfortunately, when skipping over an adjacency list we need to know k, and this fact implies that positioning on the start of a given adjacency list may require a recursive

[2]The compression speed depends essentially on W: on the WebBase graph, for example, offline recompression (i.e., both the source and the target graph are not loaded into memory) was performed at about 111 000 nodes/s when $W = 1$, 88 000 nodes/s when $W = 3$ and 54 000 nodes/s when $W = 7$; note, however, that these figures comprise both the decompression of the source and the compression of the target.

[3]These figures, however, are averaged over three data sets of 25, 50, and 100 Mpages.

18.5 Mpages, 300 Mlinks from .uk									
R	Average reference chain			Bits/node			Bits/link		
	$W=1$	$W=3$	$W=7$	$W=1$	$W=3$	$W=7$	$W=1$	$W=3$	$W=7$
∞	171.45	198.68	195.98	44.22	38.28	35.81	2.75	2.38	2.22
3	1.04	1.41	1.70	62.31	52.37	48.30	3.87	3.25	3.00
1	0.36	0.55	0.64	81.24	62.96	55.69	5.05	3.91	3.46
Tranpose									
∞	18.50	25.34	26.61	36.23	33.48	31.88	2.25	2.08	1.98
3	0.69	1.01	1.23	37.68	35.09	33.81	2.34	2.18	2.10
1	0.27	0.43	0.51	39.83	36.97	35.69	2.47	2.30	2.22
118 Mpages, 1 Glinks from WebBase									
R	Average reference chain			Bits/node			Bits/link		
	$W=1$	$W=3$	$W=7$	$W=1$	$W=3$	$W=7$	$W=1$	$W=3$	$W=7$
∞	85.27	118.56	119.65	30.99	27.79	26.57	3.59	3.22	3.08
3	0.79	1.10	1.32	38.46	33.86	32.29	4.46	3.92	3.74
1	0.28	0.43	0.51	46.63	38.80	36.02	5.40	4.49	4.17
Tranpose									
∞	27.49	30.69	31.60	27.86	25.97	24.96	3.23	3.01	2.89
3	0.76	1.09	1.31	29.20	27.40	26.75	3.38	3.17	3.10
1	0.29	0.46	0.54	31.09	29.00	28.35	3.60	3.36	3.28

Table 7: Experimental data about reference chains with $L_{\min} = 3$ and using ζ_3 for residuals. The .uk data were gathered using UbiCrawler; the WebBase data refer to the 1/2001 general crawl.

access to a number of other adjacency lists which is *not* bounded by the maximum reference count R, or by any other parameter.

To avoid this problem, we slightly change the way adjacency lists are loaded into memory: lists are considered as divided into blocks, containing J lists each (the offset array contains the bit where each such block starts). The J outdegrees are stored at the *beginning* of each block, followed by the remaining data. The advantage of this variant is that we can know the outdegree of any node without decompressing its adjacency list. This in turn solves the skipping problem completely. (As a side remark, if only sequential access is needed, the offset array needs not be loaded at all.)

6. LAZY ITERATION

Accessing a WebGraph compressed graph without references is, of course, trivial: it is just a matter of rebuilding the extra list (possibly merging intervals and residuals). This can be easily accomplished by reading all interval data, and merge the resulting sequence of integers with the residuals. Note that if you want to produce the successor list in increasing order you do not need to read actually all residuals to produce the first output, but you need to read all interval data (which, however, is usually much smaller).

Accessing a graph with (possibly heavy) referentiation is, however, a different problem, as unbounded reference chains of actual Web graphs reach more than one hundred depth levels. A trivial solution is to decode recursively each list by computing the list it references. This, however, has several disadvantages: first, a large number of memory accesses—potentially, as many as *the number of successors present in all referenced lists*. Second, a large number of temporary data structures have to be built.

WebGraph enumerates successors using *lazy iterators*; more precisely, it generates recursively a cascade of lazy iterators over the various reference lists involved in building the required one. Every iterator, at construction time, fetches all data up to the residuals (note that this data are usually very small compared to the list), but does not go farther. Then, each time the top-level iterator is required to produce a new successor, it checks whether it can fulfill the request using its local data (intervals and residuals); if it is not so, it passes the request to the iterator of the reference node.

This architecture is implemented in a very simple way, combining the powerful iterator-related features of the `fastutil` Java package, and some custom-made iterators, which implement masking of underlying iterators, merging of increasing iterators, and interval-based enumeration. We refer the reader to the API documentation for more information about these features.

Note that *no list is ever actually expanded into memory*. During the iteration, the only state kept by the recursive stack of iterators is related to intervals and blocks. This allows to iterate over very long successor lists, almost independently of the core memory available, a fact that dramatically reduces memory accesses.

Finally, since all iterators implement a method that skips several entries (and, of course, this method is implemented in constant time by interval iterators), masking can be performed very efficiently, as discarding blocks can be skipped quickly[4].

Data about access speed is provided in Table 8, where it appears clearly that for random access the speed obtained in sequential access reduces depending on the length of the average reference chain. Without lazy iterators, we would expect essentially that the speed be multiplied by the average reference-chain length, but the results are much better because of the interplay between skipping and intervalisation (indeed, the results are particularly good for the transposed graph, which is highly intervalised). Interestingly,

[4]Experiments show, indeed, that lazy iteration is *faster* than eager iteration (e.g., than simply expanding all referenced lists).

$R = \infty$ gives *better* sequential access times: in that case the main cost is memory access, and the higher compression ratio turns out to have a visible effect. Table 8 contains also data about the (small) slowdown produced by partially loading the offset array.

All data have been gathered using a Java implementation of the WebGraph framework; it is likely that a tuned C implementation would increase by a factor of about two the timings we report.

Timings given in the literature [11, 12, 10] are in the same order of magnitude of the ones reported here (e.g., version 3 of the LINK database provides a sequential and random access times are 248 ns and 336 ns, respectively, albeit with a different language and architecture). A more precise comparison would require implementing all techniques using the same language, the same compilers (and possibly virtual machines), and the same hardware.

All data have been gathered on a 512 MiB 2.4 GHz Pentium, by scanning completely the adjacency lists of a large sample of nodes. Note that sequential-access times do not depend on R, but only on W.

7. RELATED WORK

Since the Web grows constantly, and at a high speed, several papers have recently addressed the problem of compressing the Web graph. One of the first attempts at using modern compression techniques were the Connectivity Server [2], which suggested lexicographical ordering as a way to obtain a good compression using gaps, and the LINK database [11], which implemented reference compression. Hardness results about finding the best possible reference list are given in [1].

More recent attempts try to exploit more heavily the network structure of the Web, either by partitioning successor lists following hosts and popularity [12], and then using suitable codes, or using a more hierarchical view of the nodes [10]. The latter proposal, to our knowledge, implements the strongest compression techniques available before WebGraph (but see below). Reference [12] discusses also compression of other data related to the Web graph, such as URLs.

The authors of [3] present a general framework for compressing graphs that satisfy a separation condition, and apply their techniques to the Google data set (about five million edges). Their results, in bits per edge, are even better than the one presented by [10] (albeit worse than WebGraph), but an actual comparison is very difficult, because the experimental data set is quite small, and it is hard to see whether the compression ratios will extend to larger Web graphs. Access times are very good (in fact, the best so far), but this kind of data is even more sensitive to scaling issues, as they are strongly dependent on low-level caching, hardware details, and implementation.

We do not know of any attempts to use lazy iteration to enumerate successors without decompressing adjacency lists in main memory.

8. CONCLUSIONS

In this paper, we presented new techniques for compressing the Web graph and its transposed. The number of bits per link obtained is a major advance with respect to all methods we are aware of. Moreover, the format is very flexible, and can be tuned so to fit different kinds of CPUs and memory sizes.

All software and data sets described in the experimental part are freely available from the WebGraph home page (`http://webgraph.dsi.unimi.it/`), where one can also get large data sets. The Java part of the WebGraph framework is fully documented using the JavaDoc documentation system.

There is, of course, much work still to be done. Presently, the most interesting challenge is to find a simple way to get the optimal compression from a given sequence of successors, and a given reference list. This is more involved than it may seem at first sight, as, for instance, it could be interesting to store redundantly nodes both as intervals and as references just to get larger intervals.

Another interesting question is exploring the possible interplay between different compression parameters. Since WebGraph allows a great freedom in choosing codes, it may happen, that, say, a very large window W works very well if one chooses the right code (say, γ instead of unary) for the reference.

Finally, WebGraph does not provide any way to store efficiently the URLs of a Web graph. In the future, for instance, we plan to implement a new minimal perfect hash class that also stores the URLs themselves, and acts as a two-way connection between URLs and nodes.

9. REFERENCES

[1] Micah Adler and Michael Mitzenmacher. Towards compressing Web graphs. In *Proc. of the IEEE Data Compression Conference*, pages 203–212, 2001.

[2] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The Connectivity Server: Fast access to linkage information on the Web. In *Proceedings of the Seventh International World–Wide Web Conference*, pages 469–477, Brisbane, Australia, 1998.

[3] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *Proceedings of the Fourteenth Annual ACM–SIAM Symposium on Discrete Algorithms (SODA−03)*, pages 579–688, New York, 2003. ACM Press.

[4] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed Web crawler. In *Proc. AusWeb02. The Eighth Australian World Wide Web Conference*, 2002. To appear in *Software: Practice & Experience*.

[5] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: Scalability and fault-tolerance issues. In *Poster Proc. of Eleventh International World Wide Web Conference*, Honolulu, USA, 2002.

[6] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework II: Codes for the World–Wide Web. Technical Report 294-03, Università di Milano, Dipartimento di Scienze dell'Informazione, 2003. Submitted for publication. Available at `http://webgraph.dsi.unimi.it/`.

[7] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of Web pages. In *Proc. of WWW9*, Amsterdam, The Netherlands, 2000.

[8] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, September 1999.

[9] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford Digital Library Technologies Project, Stanford University, Stanford, CA, USA, 1998.

[10] Sriram Raghavan and Hector Garcia-Molina. Representing Web graphs. In *Proc. of the IEEE Intl. Conference on Data Engineering*, 2003.

18.5 Mpages, 300 Mlinks from .uk									
R	Graph size (MiB)	Offset-array size (MiB)				Link access time (ns)			
		seq.	$J=1$	$J=2$	$J=4$	seq.	$J=1$	$J=2$	$J=4$
∞	79.0					198	31 237	35 752	43 699
3	106.6	–	141.3	70.7	35.3	206	611	753	886
1	122.9					233	442	491	605
Transpose									
∞	70.3					150	2 382	2 873	2 961
3	74.6	–	141.3	70.7	35.3	171	342	424	516
1	78.8					183	234	312	374

Table 8: Experimental data about access time, obtained on 512 MiB 2.4 GHz Pentium for a 18.5 Mpages snapshot of the .uk domain.

[11] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener. The LINK database: Fast access to graphs of the Web. Research Report 175, Compaq Systems Research Center, Palo Alto, CA, 2001.

[12] Torsten Suel and Jun Yuan. Compressing the graph structure of the Web. In *Proc. of the IEEE Data Compression Conference*, pages 213–222, 2001.