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.

Copyright is held by the author/owner(s).
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).

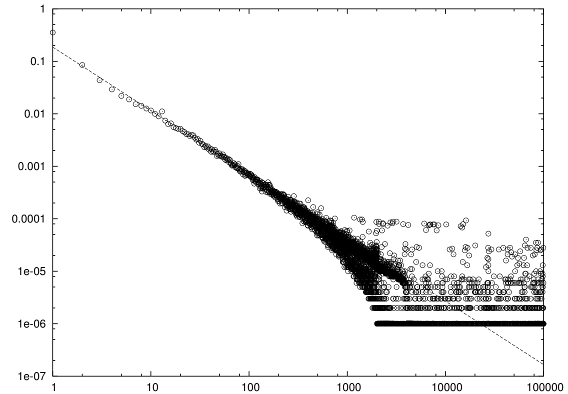

Figure 1: Distribution of gaps in a 18.5 Mpages snapshot of the `.uk` domain. The scale is logarithmic on both axes, and the line displays a power law with exponent 1.21.

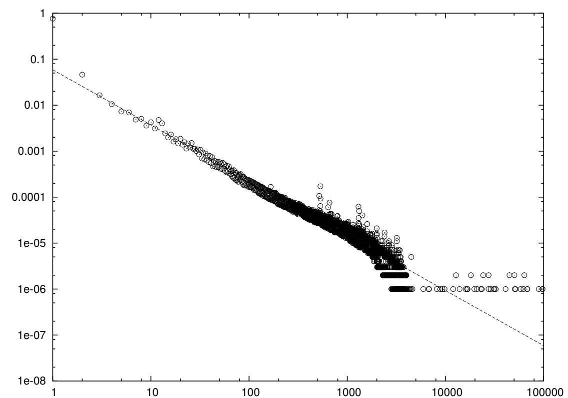

Figure 2: Distribution of gaps in the transpose of a 18.5 Mpages snapshot of the `.uk` domain. The scale is logarithmic on both axes, and the line displays a power law with exponent 1.20 (modulo a scaling factor).

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.