Efficient and Simple Encodings for the Web Graph

Jean-Loup Guillaume, Matthieu Latapy, Laurent Viennot

Efficient and Simple Encodings for the Web Graph.

Jean-Loup Guillaume [1], **Matthieu Latapy** [1,2]
and Laurent Viennot [2]

Abstract: In this paper, we propose a set of simple and efficient methods based on standard, free and widely available tools, to store and manipulate large sets of URLs and large parts of the Web graph. Our aim is both to store efficiently the URLs list and the graph in order to manage all the computations in a computer central memory. We also want to make the conversion between URLs and their identifiers as fast as possible, and to obtain all the successors of an URL in the Web graph efficiently. The methods we propose make it possible to obtain a good compromise between these two challenges, and make it possible to manipulate large parts of the Web graph.

Keywords: Web graph, Web links, URLs, Compression.

Approximate words count: 3500.

1. Introduction.

One can view the Web as a graph whose vertices are Web pages, and edges are hyperlinks from one page to another. Understanding the structure of this graph is a key challenge for many important present and future applications. Information retrieval, optimized crawling and enhanced browsing are some of them. The first step to study the Web graph is to be able to store and manipulate it efficiently, both in space and in time terms. The key element of this encoding is to associate a unique identifier to each URL which will then be used to encode the graph.

URLs are more than 70 bytes long on average and each vertex has an average outdegree at least seven, depending on the considered domain (from 7.2 in [8] to 11.2

[1]LIAFA, Université Paris 7, 2, place Jussieu, 75005 Paris, France. (guillaume,latapy)@liafa.jussieu.fr, +33 (0) 1 44 27 28 37

[2]Projet Hipercom, INRIA Rocquencourt, F-78153 Le Chesnay (France). Laurent.Viennot@inria.fr, +33 (0) 1 39 63 52 25

in [1] and 11.57 for the data we used in our experiments). Encoding a one million vertices subgraph of the Web graph without any compression would therefore need more than 100 MB of memory. When one is concerned with the Web graph, it is important to deal with much bigger graphs, classically several hundreds of millions vertices. Therefore, the efficient encoding of the graph becomes a crucial issue. The challenge is then to find a good balance between space and time requirements.

Until now, the main work concerning graph encoding is the Connectivity Server presented in [2]. This server maintains the graph in memory and is able to compute the neighborhood of one or more vertices. In the first version of the server, the graph is stored as an array of adjacency lists, describing the successors and predecessors of each vertex. The URLs are compressed using a delta compressor: one URL is stored using only the differences from the previous one in the list. The second [3] and present version [10] of the Connectivity Server have significantly improved the compression rate for both links and URLs. The space needed to store a link has been reduced from 8 to 1.7 bytes in average, and the space needed to store a URL has been reduced from 16 to 10 bytes in average. Notice however that a full description of the method is available only for the first version of the server [2], the newer (and more efficient) ones being only shortly described in [3, 10].

Our aim is to provide an efficient and simple solution to the problem of encoding large sets of URLs and large parts of the Web graph using only standard, free and widely available tools, namely `sort`, `gzip` and `bzip`. The `gzip` tool is described in [5, 6] and `bzip` algorithm in [4]. We tested our methods on a 8 millions vertices and 55.5 millions links crawl performed inside the "`.fr`" domain in June 2001. We used the crawler designed by Sebastien Ailleret, available at the following URL:
`http://pauillac.inria.fr/~ailleret/prog/larbin/index-eng.html`
Our set of data itself is available at:
`http://hipercom.inria.fr/~viennot/webgraph/`
It has been obtained by a breadth-first crawl from a significant set of URLs. See the URL above for more details on these data. Although it may be considered as relatively

small, this set of data is representative of the Web graph since it is consistent with the known statistics (in particular in terms of in- and out-degree distribution [1, 3], and for the average length of URLs, which are the most important parameters for our study).

All the experiments have been made on a Compaq™ Workstation AP 550, with a 800 MHz Pentium™ III processor, with 1 GB memory and a Linux 2.4.9 kernel. We obtained an encoding of each URL in 6.54 bytes on average with a conversion between URLs and identifiers (in both directions) in about 2 ms. One-way links can also be compressed to 1.6 byte on average with immediate access (around 20 μs), which can be improved to 1 byte if one allows slower access.

We describe in Section 2 our method to associate a unique identifier to each URL, based on the lexicographical order. We show how to compress the URLs set and how to obtain fast conversion between URLs and identifiers. In Section 3, we notice some properties on the graph itself, concerning a notion of distance between vertices and their successors. These properties explain the good results obtained when we compress the graph. Two different and opposite approachs are discussed concerning the compression: one of them optimizes space use, and the other one optimizes access time.

2. URLs Encoding.

Given a large set of URLs, we want to associate a unique identifier (an integer) to each URL, and to provide a function which can make the mapping between identifiers and URLs. A simple idea consists in sorting all the URLs lexicographically. Then a URL identifier is its position in the set of sorted URLs. We will see that this choice for an identifier makes it possible to obtain efficient encoding.

Let us consider a file containing a (large) set of URLs obtained from a crawl. First notice that sorting this file improves its compression since it increases the local redundancy of the data: we obtained an average of 7.27 bytes by URL before sorting and an average of 5.55 bytes after sorting (see Table 1). This space requirement is

very low, and it may be considered as a lower bound. Indeed, using this compression method is very inefficient in terms of lookup time, since when one converts a URL into its identifier and conversely, one has to uncompress the entire file. On the other hand, random access compression schemes exist [7, 9], but their compression rate are much lower, too much for our problem. Notice than one can also use `bzip` [4] instead of `gzip` to obtain better compression rates (but paying it by a compression and expansion slowdown). However, we used `gzip` in our experiments because it provides faster compression and expansion routines, and is more easily usable, through the `zlib` library for instance.

2.1. **Encoding by `gzip`ped blocks.** To avoid the need of uncompressing the entire list of URLs, we split the file into blocks and compress independently each of them. We also know the first URL of each block, together with its identifier. We save this way a large amount of time since only one block has to be uncompressed to achieve the mapping. Moreover, since the URLs are sorted, the ones which share long common prefixes are in the same block, and so we do not damage the compression rate too much (in some cases, we even obtain a better compression rate than when one compresses the entire file).

Experimentally, the average size for a compressed URL does not significantly increases as long as blocks length stays over one thousand URLs. In this case, URL average size is 5.62 bytes long. With blocks of one hundred URLs, the average size grows up to 6.43 bytes long. Notice that the method can be improved by taking blocks of different sizes, depending on the local redundancy of the URLs list. We did not use this improvement in the results presented here, which have therefore been realized with blocks of constant length.

One can then convert a URL into an identifier as follows:

1. Find the block which contains the URL to convert: use a dichotomic search based on the knowledge of the first URL of each block (either because we kept a list of those URLs, or by uncompressing the first line of each concerned block, which have a constant cost).

Encoding	total size (8 millions URLs)	Average size/URL
Text	568733818 bytes	69.24 bytes
`bzip`	36605478 bytes	4.45 bytes
`gzip`	45263569 bytes	5.55 bytes

TABLE 1. Average URL size according to coding format.

2. Uncompress the block.
3. Find the identifier of the URL inside the (uncompressed) block: use a linear search in the list (we cannot avoid this linear search since all the URLs do not have the same length).

This conversion scheme is summarized in Table 2.

Conversely, one can convert an identifier to a URL as follows:

1. Find the block which contains the identifier to convert: since all the blocks contains the same number of URLs, the block number is given by $\frac{\text{Identifier}}{\text{BlocksLength}}$.
2. Uncompress the block.
3. Find the URL in the (uncompressed) block: it is nothing but the line number $\text{Identifier} - \text{BlocksLength} \cdot \text{BlockNumber}$ in block. Again, we need to use a linear search in the list.

This conversion is summarized in Table 2.

	URL to identifier	identifier to URL
First step	O(log(number of blocks))	O(1)
Second step	O(blocks length)	O(blocks length)
Third step	O(blocks length)	O(blocks length)

TABLE 2. URL to identifier and identifier to URL mapping costs, when all the URLs do not have the same length inside a block.

Notice that, because of the linear search in a block (Step 3 of each conversion), it is important that each block is short enough. However, this can be improved by the

use of a fixed length for all the URLs in each block. This is what we will present in the following subsection.

2.2. **Fixed URLs length.** To improve the lookup time, we add at the end of all the URLs in a given block as many occurrences of a special character as necessary to make it as long as the longest URL in the block. In each block, the fixed length is then the length of the longest URL. Therefore, the third point of the URL to identifier conversion becomes a dichotomic search in the block, and the third point of the identifier to URL conversion can be done in constant time since the URL is at position $\text{UrlsLength} \cdot (\text{Identifier} - \text{BlocksLength} \cdot \text{BlockNumber})$ in the block. This improvement is summarized in Table 3.

	URL to identifier	identifier to URL
First step	O(log(number of blocks))	O(1)
Second step	O(blocks length)	O(blocks length)
Third step	O(log(blocks length))	O(1)

TABLE 3. URL to identifier and identifier to URL mapping costs, when all the URLs have the same length inside a block.

Notice that this optimization must be done carefully to ensure both a good compression of the URLs and a fast expansion (Step 2). If the blocks size is too low, compression rate will be naturally low. On the opposite, if the size if too important, the probability that a very long URL lies in the file will increase, adding a lot of unused character, which are going to increase the average URL size. Expansion time is linear with respect to the blocks length, so we must use as small blocks as possible to get fast mapping. Using median blocks length will result in very good compression rate but median expansion speed. Results showing these phenomena can be found in Figure 1.

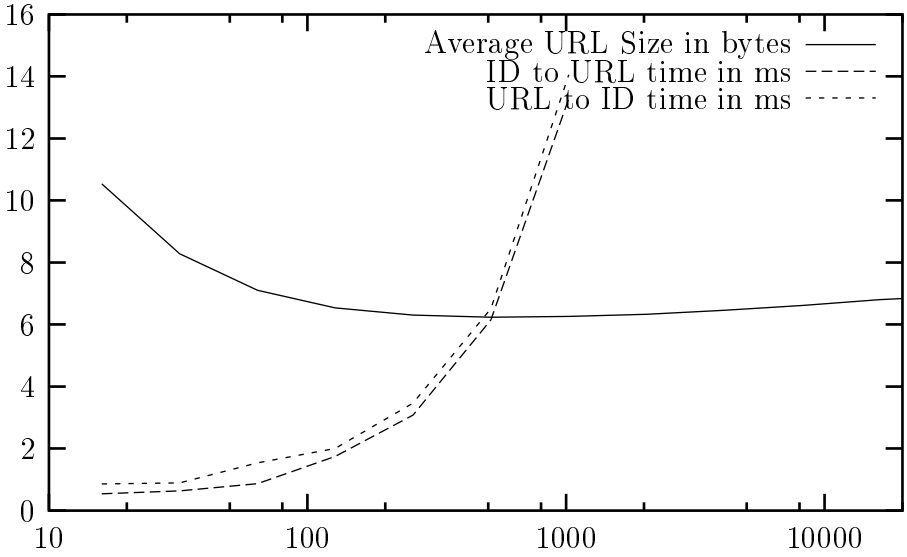

FIGURE 1. Average URL size and conversion times with respect to the size of the considered blocks, using fixed-length URLs.

In conclusion, we obtained a coding of the URLs in 6.54 bytes in average, with conversion between URLs and their identifiers in about 2 ms (in both directions), using only simple, free and widely available tools (`sort` and `gzip`). This coding associates to each URL its position in the entire list with respect to the lexicographic order, and we show how one can compute the correspondence efficiently. We will now see how this encoding can be used to represent large parts of the Web graph.

3. Graph Encodings.

As soon as the mapping between URLs and identifiers is defined, we can try to compress all links as much as possible. A link is defined by a couple of integers, each of them being the identifier of a URL as defined in Section 2. The graph is then stored in a file such that line number k contains the identifiers of all the successors of vertex k (in a textual form). Using `bzip` to compress this file, we obtain a very compact encoding: 0.8 byte by link on average. If one uses `gzip` instead of `bzip`, the average size of each link grows up to 0.83 byte on average. Again, these values may be considered as lower bounds for the space needed to represent a link.

In this section, we will propose two methods to encode the links of the Web graph. The first one is a simple extension of the `gzip`ped blocks method used in the previous section. It gives high compression rates, which can be understood as a consequence of a strong locality of the links we will discuss. In order to improve the access time to the successors of a vertex, which is very important to be able to make statistics and run algorithms on the graph, we propose a second method which achieve this goal but still allows high compression rates. Notice that the techniques we present in this section can be used to encode the reverse links (given an URL, which pages do contain a link to this URL). The performances would be similar.

3.1. **Encoding by `gzip`ped blocks.** Using the same method as in Section 2.1, we can split the file representing the graph into blocks and then compress the blocks. In order to find the successors of a vertex, one has to uncompress the block containing the vertex in concern. Once this has been done, vertex successors have to be found. Depending on how successors are coded, two different searching methods can be used. If successors lists have variable length, one has to read the block linearly from the beginning to the right successors list. On the other hand, if successors have fixed length (this can be done in the same way as for the URLs) then the successors list can be found directly. Notice that in both cases, since most of the lookup time is spent in the block expansion, there is no real time difference between getting one successor of a vertex, or the entire list of its successors. Average lookup time and link average size can be found in Figure **??**. One can obtain an encoding of each link in 1.24 byte in average with a lookup time of 0.45 ms, using 32 lines blocks. Table 4 present the results when block size change.

However, most of the operations made on the graph concern the exploration of successors or predecessors of vertices (during breadth-first search for instance). In this case, successors lookup time becomes a crucial parameter, and block compression method should be improved in terms of time. We are going to present another compression method which uses a strong property of the Web graph, the locality, to improve lookup time. ’

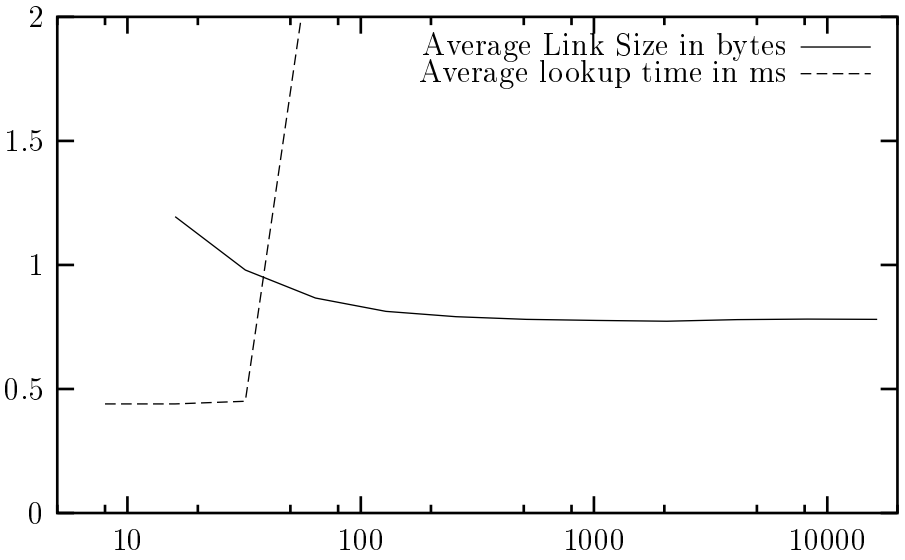

FIGURE 2. Average link size and average lookup time with respect to the size of the considered blocks.

3.2. **Locality.** The high compression rates we obtained when we encoded the graph using `gzip` can be understood as a consequence of a strong property of the links. Let us define the *distance* between two URLs as the (signed) difference between their identifiers, and the *length* of a link between two URLs as the distance between these two URLs. Now, let us consider the distances distribution. This distribution follows a power law: the probability for the distance between two given vertices to be i is proportional to $i^{-\tau}$. In our case the exponent τ is about 1.16. See Figure 3.

One may want to use this locality to improve both compression rate and access time by encoding the graph in a file as follows: the k-th line of the graph contains the successors of URL number k, encoded by their distance to k. We can then use the same technique of `gzip`ped blocks encoding to manipulate the graph. We tried this method, but we obtained lower compression rates than the ones presented in the previous subsection. However, this encoding may be used to improve lookup time, without damaging compression rate too much, as explained in the following subsection.

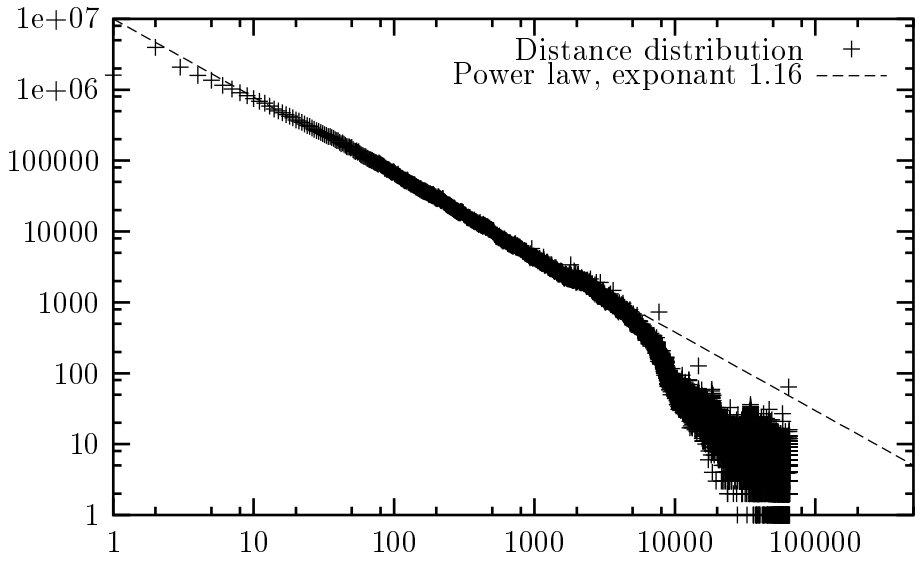

FIGURE 3. Distance distribution between vertices and their successors.

3.3. **Access time improvement.** Our experiments show that 68 percent of the URLs which are linked together are at distance between -255 and 255. We call these links *short* links. They can be encoded on 1 byte, plus 1 bit for the sign of the difference. Moreover, we need one more bit to distinguish short links from long ones (the long links are encoding using 3 bytes, since we are considering a 8 millions vertices graph). This scheme allows us to encode a link using 1.89 byte on average. Going further, one can distinguish short (68 percent of the links, each encoded on 1 byte), medium (26.75 percent of the links, encoded on 2 bytes) and long (5.25 percent of the links, encoded on 3 bytes) links. We therefore use one bit per link to give the sign of the distance, and a prefix to know the type of the link (0 for short links, 10 for medium links and 11 for long links). This way, a link can be stored using 1.66 byte on average.

Moreover, the distance distribution encourages us to use Huffman compression of the distances. However, our experiments show that it is better not to compress long links using this method, and to restrict it to short links. We obtained this way an

improvement of 1 bit on average, which brings us to 1.54 byte by link. Our results are summarized in Table 4.

4. Conclusion.

We described in this paper a simple and efficient method to encode large sets of URLs and large parts of the Web graph. We gave a way to compute the position of a URL in the sorted list of all the considered URLs, and conversely, which makes it possible to manipulate large data sets in RAM, avoiding disk usage. Our `gzip`ped blocks method makes it possible to store 400 millions of URLs and the 4.6 billions links between them in 8 GB of memory space. Using this encoding, the conversion between identifiers and URLs takes around 2 ms on our computer, in both directions, and finding all the successors of a given URL takes around 0.5 ms. We can improve the link lookup to around 20 μs by using the second method we proposed, but with an increase of the space requirements.

We therefore obtained results which are comparable to the best results known in the literature, using only standard, free, and widely available tools like `sort`, `gzip` and `bzip`. Notice that the good performances of our method rely on the performances of these tools, which have the advantage of being strongly optimized.

Our work can be improved in many directions. We discussed some of them in the paper, for example the use of pieces of files of different sizes (depending on the local redundancy of the URLs list). Another idea is to try to increase the locality and the redundancy of the URLs, for example by reversing the sites names. This may reduce the distances between pages of sites which belong to a same sub-domain. There are also many parameters which depend on the priority of time or space saving, itself depending on the application. However, the optimization of memory requirements makes it possible to store the entire data in RAM, reducing disk access, and therefore is also important to improve computing time. This is why we gave priority to the optimization of space requirements, except when a big improvement in speed can be obtained.

	Average link size	Average lookup time for all the successors
identifiers	8 bytes	–
`gzip`ped identifiers	0.83 byte	–
distances	4.16 bytes	–
`gzip`ped distances	1.1 byte	–
`gzip`ped identifiers, blocks of 8 lines	1.61 byte	0.44 ms
`gzip`ped identifiers, blocks of 16 lines	1.36 byte	0.44 ms
`gzip`ped identifiers, blocks of 32 lines	1.24 byte	0.45 ms
`gzip`ped identifiers, blocks of 64 lines	1.20 byte	2.395 ms
`gzip`ped identifiers, blocks of 128 lines	1.21 byte	5.694 ms
`gzip`ped identifiers, blocks of 256 lines	1.26 byte	16.866 ms
short, long links	1.89 byte	20 μs
short, medium, long links	1.66 byte	20 μs
short (Hufmann), medium, long links	1.54 byte	20 μs

TABLE 4. The average space needed to store one link, depending on the method used. The first four lines are just here to serve as references, since they imply either a very low compression ratio, or very slow elementary operations.

An important direction for further work is to find an encoding of the graph which would allow a faster access to the successors list of a given URL. This is a key element for the study of the structure of the graph (cliques, bipartite subgraphs, connected components, and others). It seems that the use of standard compression tools is not the best method to achieve this, at least in terms of access time optimization.

References

1. R. Albert, H. Jeong, and A.-L. Barabasi, *Diameter of the world wide web*, Nature **401** (1999), 130–131.
2. Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian, *The Connectivity Server: fast access to linkage information on the Web*, WWW7 / Computer Networks and ISDN Systems **30** (1998), no. 1–7, 469–477.
3. A. Z. Broder, S. R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener, *Graph structure in the web*, WWW9 / Computer Networks **33** (2000), no. 1-6, 309–320.
4. M. Burrows and D. J. Wheeler, *A block-sorting lossless data compression algorithm.*, Tech. Report 124, 1994.
5. P. Deutsch, *Deflate compressed data format specification version 1.3*, Aladdin Enterprises, May 1996, RFC 1951.
6. ———, *Gzip file format specification version 4.3*, Aladdin Enterprises, May 1996, RFC 1952.
7. Jirí Dvorsky, *Text compression with random access.*
8. J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins, *The Web as a graph: Measurements, models, and methods*, Proc. 5th Annual Int. Conf. Computing and Combinatorics, COCOON (T. Asano, H. Imai, D. T. Lee, S. Nakano, and T. Tokuyama, eds.), no. 1627, Springer-Verlag, 1999.
9. Haris Lekatsas and Wayne Wolf, *Random access decompression using binary arithmetic coding*, Data Compression Conference, 1999, pp. 306–315.
10. Rajiv Wickremesinghe (Duke University), Raymie Stata, Janet Wiener, et al., *Link compression in the connectivity server*, Tech. report, Compaq Systems Research Center, Summer, 2000.