Internet Mathematics Vol. 6, No. 3: 257–283

Permuting Web and Social Graphs

Paolo Boldi, Massimo Santini, and Sebastiano Vigna

Abstract. Since the first investigations on web-graph compression, it has been clear that the ordering of the nodes of a web graph has a fundamental influence on the compression rate (usually expressed as the number of bits per link). The authors of the LINK database [Randall et al. 02], for instance, investigated three different approaches: an *extrinsic* ordering (URL ordering) and two *intrinsic* orderings based on the rows of the adjacency matrix (lexicographic and Gray code); they concluded that URL ordering has many advantages in spite of a small penalty in compression. In this paper we approach this issue in a more systematic way, testing some known orderings and proposing some new ones. Our experiments are made in the WebGraph framework [Boldi and Vigna 04], and show that the compression technique and the structure of the graph can produce significantly different results. In particular, we show that for a *transposed* web graph, URL ordering is significantly less effective, and that some new mixed orderings combining host information and Gray/lexicographic orderings outperform all previous methods: in some large transposed graphs they yield the quite incredible compression rate of 1 bit per link. We experiment with these simple ideas on some nonweb social networks and obtain results that are extremely promising and are very close to those recently achieved using shingle orderings and backlink compression schemes [Chierichetti et al. 09].

1. Introduction

A web graph [Kumar et al. 00] is a directed graph whose nodes correspond to URLs, with an arc from x to y whenever the page denoted by x contains a hyperlink toward the page denoted by y; more loosely, the same term is sometimes

1542-7951/09 $0.50 per page

used for the undirected version of the graph when arc direction is not relevant. Web graphs are a huge source of information, and they contain precious data that find applications in ranking, community discovery, and more. In many cases, the results obtained and the techniques applied in relation to a web graph are also appropriate for the larger realm of *social networks*, of which a web graph is only a special case; while the first studies on web-graph compression date back to the early 2000s, very large nonweb social graphs have been made available only more recently, triggering research about their compressibility, as had happened in the case of web graphs a decade earlier.

Indeed, one nontrivial practical issue in dealing with such graphs is their size: a typical web graph or real-world social network contains millions, sometimes billions, of nodes, and although sparse, its adjacency matrix is way too big to fit in main memory, even on large computers. To overcome this technical difficulty, one can access the graph from external memory, which, however, requires that one design special offline algorithms even for the most basic problems (e.g., finding connected components or computing shortest paths); alternatively, one can try to compress the adjacency matrix so that it can be loaded into memory and still be directly accessed without decompressing it (or decompressing it only partially, on demand, and efficiently).

The latter approach, which can be referred to as *web-graph compression*, can be traced back to the LINK database [Randall et al. 02]; more recently, it led to the development of the WebGraph framework [Boldi and Vigna 04], which still provides some of the best practical compression-versus-speed tradeoffs.

Most web-graph compression algorithms strongly rely on properties that are satisfied by typical web graphs; in particular, the key properties that are exploited to compress the graph adjacency structure are locality and similarity. *Locality* means that most links from page x point to pages of the same host as x (and often share with x a long path prefix).[1] *Similarity* means that pages that are from the same host tend to have many links in common (this property is becoming more and more frequent with the widespread use of templates and generated content).

The fact that most compression algorithms make use of these (and similar) properties explains why such algorithms are so sensitive to the way nodes are ordered. A technique that works incredibly well, and was adopted already in the LINK database [Randall et al. 02], consists in sorting nodes lexicographically by URL (the first node is the one that corresponds to the lexicographically first URL, and so on). In this way, successor lists contain by locality URLs that

[1] According to the RFC 1738, the *host* is the string between the second and the third slash in an absolute URL.

are assigned close numbers, and *gap encoding*, a standard technique borrowed by inverted-index construction, makes it possible to store each successor using a small number of bits.[2] This solution is usually considered good enough for all practical purposes, and has the extra advantage that even the URL list can be compressed very efficiently via prefix omission [Bharat et al. 98]. Analogous techniques, which use additional information besides the Web graph itself, are called *extrinsic*.

It is natural to wonder whether there is an alternative way of finding a "good ordering" of the nodes that allows one to obtain the same (or perhaps better) compression rate without having to rely on URLs; this is especially urgent for social networks, where nodes do not themselves correspond to URLs.[3] A general way to approach this problem may be the following: take the graph with some ordering of its n nodes, and let A be the corresponding adjacency matrix; now, based on A, find some permutation π of its rows and columns such that when applied to A, it produces a new matrix A' in which two rows that are close to each other (i.e., appear consecutively, or almost consecutively) are similar (i.e., they contain 1's more or less in the same positions).

Finding a good permutation π is itself an interesting problem; in [Randall et al. 02], the authors propose choosing the permutation π that would sort the rows of A in lexicographic order. This is an instance of a more general approach: fix some total ordering $\prec$ on the set of n-bit vectors (e.g., the lexicographic ordering), and let π be the permutation that sorts the rows of A according to $\prec$.[4] Observe that the rows of A' are *not* $\prec$-ordered, because the permutation π is applied to *both* rows and columns. These approaches are called *intrinsic*, since they do not rely on external information (such as the URLs of each node).[5]

Another possible solution to the same problem, already briefly mentioned in [Randall et al. 02], consists in letting $\prec$ be a Gray ordering, that is, an ordering where adjacent vectors differ by exactly one bit. Although this solution may sound promising, one should carefully determine an efficient algorithm that finds the sorting permutation π with respect to (some) Gray ordering.

In this paper we explore experimentally, using the WebGraph framework, the improvements in compression due to permutations. Besides the classical permu-

[2] In gap encoding, instead of storing $x_0, x_1, x_2, \dots$, we store, using a variable-length bit encoding, x_0, $x_1 - x_0$, $x_2 - x_1$, and so on.

[3] We note that the same approach has been shown to be fruitful in the compression of inverted indices; see, for instance, [Blandford and Blelloch 02, Shieh et al. 03, Silvestri 07, Blanco and Barreiro 05].

[4] In this description we are ignoring the problem that π is not unique if A contains the same row many times.

[5] Of course, it is possible to devise intrinsic methods that do not necessarily depend on some ordering; see, for instance, [Blandford et al. 03].

tations described above, we propose two new permutations based on the Gray ordering. However, we restrict the permutation to rows of the same host. Moreover, for the first time we provide experimental data on the *transposed* graph (i.e., the graph obtained by reversing the direction of all arcs), showing that intrinsic permutations provide a dramatic increase in compression, in contrast to what happens in the standard case.

Some of the methods described in the paper are totally intrinsic and can thus be applied to any graph; other techniques, although not fully intrinsic, can be adapted to work also without any additional information about the nodes. It is hence natural to wonder whether they can be fruitfully exploited on other social networks. Social network compression has been recently discussed in [Chierichetti et al. 09], where the authors proposed two intrinsic ordering heuristics based on shingles, and also study a compression scheme called *backlink (BL) compression* that improves on the WebGraph format, adding the idea of reciprocity. In the experimental part of this paper, we show how our methods behave when applied to nonweb social networks; our results (still based on the WebGraph framework) are promising, and they produce compression rates that are comparable to those shown in [Chierichetti et al. 09], while providing fast access.

Before discussing our approach and results, though, we believe that it is worthwhile taking some time to discuss the current state of the art in web-graph compression: this is the content of the next section.

2. Web Graph Compression: A Primer

As briefly explained in the introduction, when large graphs (and, more generally, large amounts of data) are to be processed and their size is such that they cannot fit in main memory directly, two opposing approaches can be adopted. One possibility is to assume that the graph is never actually loaded into memory, but is rather read (possibly more than once) in a streaming fashion from external storage using a small amount of memory. The second is to rely on some graph-compression techniques that allow one to reduce the graph space usage enough to be able to load it in memory.

As a first, general, remark, let us distinguish between *compressed data structures* and *compression schemes*: the latter are evaluated only on the basis of their space efficiency (how much space the compressed data occupy), whereas the former require that accessing the data can be done efficiently and without (fully) decompressing it. There is, of course, an information-theoretic lower bound on the performance of compression schemes (and a fortiori of compressed data structures), given by a simple counting argument: any compression scheme for objects of a universe Ω cannot use less than $\log_2 |\Omega|$ bits on average; in some cases, it is actually possible to design compression schemes that achieve this lower bound, and are hence space-optimal, at least in the average case. For directed labeled graphs, the adjacency matrix is optimal (since there are 2^{n^2} directed labeled graphs). Turán, in one of the early papers on the subject [Turán 84], described a representation for planar graphs and posed the problem of encoding general *unlabeled* graphs (i.e., graphs that should be considered up to automorphism); a solution was found a few years later in [Naor 90]. Albeit interesting, this kind of result is of limited practical import for two reasons: no efficient method is usually provided to access the data without decompressing it entirely, and moreover, in real applications, the empirical distribution of the objects is far from being uniform—in many cases, a scheme that works poorly in the worst or average case can be preferable if it is able to compress typical instances efficiently.

On the other hand, compressed data structures are in a sense the empirical counterpart of *succinct* data structures (introduced in [Jacobson 89]), which store data using a number of bits equal to the information-theoretic lower bound, providing access time asymptotically equivalent to a standard data structure; in particular, a compressed data structure for a graph must provide very fast amortized random access to an arc (link), say on the order of a few hundred nanoseconds. More precisely, the kind of compressed data structures for graphs that are the topic of our discussion should feature the following (somewhat loose) properties:

- they must exhibit good compression ratios (possibly below the information-theoretic lower bound) at least for the kind of graphs one wants to deal with (in our case, web graphs, social networks, and the like);
- they should possibly guarantee good compression on average, not far from the lower bound;
- they should be endowed with access primitives that are almost as efficient as those of a noncompressed graph structure;
- they can be *coordinate-free*: by this we mean that the data structure yields approximately the same compression rate *independently of the specific permutation of the graph it is given to process.*

Which specific access primitives are provided depends on the way compression is achieved as well as on its intended use. Broadly speaking, we can distinguish between *sequential* access (scan through all the arcs, with guarantees that the arcs going out of each node are provided consecutively), *random* access (given x, provide the list of successors of x), or *direct* access (given x and y, tell whether there is an arc from x to y). In some cases, random or direct access to the transposed graph might also be made available. Note that we are not interested in dynamic data structures: we are assuming that the graphs under consideration are static and immutable; this implies, in particular, that we are disregarding the time taken for construction (compression) but we are assuming that compression can be performed using a reasonable amount of time and space, precisely because there is no other way to modify the graph than to reconstruct everything from scratch. Unfortunately, compression time and space are not usually reported in papers about web-graph compression. The rationale behind this choice is that since such graphs will be stored once and read many times, construction time is not very relevant. On the other hand, apart from the considerations above, it is necessary that the compression techniques be very scalable, since the Web graph is growing every day. The lack of experiments beyond, say, 100 million nodes may be a sign that the technique described does not really scale up.

The features of the links of a web graph that are usually quoted as *locality* and *similarity* (described in the introduction) were originally exploited by the LINK database [Randall et al. 02]; given that the nodes are sorted lexicographically by URL, locality guarantees that most links are between nodes close to each other in the order, and similarity suggests that nearby nodes have similar successor lists. The authors of the LINK database write each successor list as a sequence of gaps (the smallest successor of node x is written as a difference from x, and each of the remaining successors, in increasing order, is written as a difference from the previous one). Locality implies that the gaps written in this way are small, and using some suitable universal code for the integers, they can be written compactly; in [Randall et al. 02], a nybble code is used to write gaps (binary values are written as sequences of three-bit groups, plus one continuation bit). The reason behind this choice is that although nybble codes are outperformed by Huffman, they are much faster to decompress. As explained in [Asano et al. 03], a posteriori, the choice of nybble codes is optimal (at least for gaps between successors) among all the universal codes of the same family (the so-called k-bit variable-length code, where $k = 3$ for nybble codes). Indeed, empirical analysis performed by [Asano et al. 03] on web graphs shows that such gaps exhibit a power-law distribution with exponent around 4/3, and the optimal k for this exponent is 3.[6]

[6] In the same work, the authors show that the first gap, i.e., the (absolute value of the) difference between the source and the smallest successor, has also a power-law distribution but with a different exponent (7/6 instead of 4/3), which would suggest using a six-bit variable-length code instead.

Besides its clever usage of locality, [Randall et al. 02] also analyzes the idea of storing similar successor lists by reference: if a list of successors is sufficiently similar to one of those that appeared immediately before it, only the symmetric difference between the two successor sets is written: a bit list records which successors of the referenced list are actually used. The idea of using similarity was also explored independently in [Adler and Mitzenmacher 01], which also presents some negative results about the complexity of finding the "best possible" node to copy from, and suggests a number of possible practical improvements over the bit-list idea, in particular that of run-length encoding such a list, which is similar to the inclusion–exclusion blocks used by WebGraph [Boldi and Vigna 04].

The results of [Randall et al. 02] were obtained on large data sets, on which was obtained a compression rate of about 5.5 bits/link: this value includes the offset data structure that allows for random access, but it is not clear from the paper how much one can gain in space if only sequential access is required. Sequential and random access require about 220 and 300 ns/link, respectively (on a Compaq with 2-GHz CPUs).

Apart from [Randall et al. 02], there was a flurry of activity about the same problem in the early 2000s, some of which delivered ideas that were later used in other frameworks. In 2002, [Guillaume et al. 02] considered two solutions, both based on the idea of grouping the successor lists into blocks and then compressing each block separately, either by gzipping it or by writing each successor as a difference from the source (once more, exploiting locality albeit in a slightly different way); they did not use a standard universal code for writing the differences, but rather employed three different encodings depending on the size of the integer to be written. The best compression result they obtained is about 13 bits/link, with access time of about 5.1 μs/link (on a Compaq with an 800-MHz Pentium III). Observe that [Guillaume et al. 02] decided to release the data set on which their experiments were run: they were the first to do so, to the best of our knowledge.[7]

The authors of [Asano et al. 03] tried to learn from the lessons of [Randall et al. 02] and [Guillaume et al. 02], and also introduced new ideas: in particular, they observe that often successors are consecutive to each other and propose

[7] It is worth spending some words here about the intrinsic difficulty in comparing different compression techniques proposed and discussed in the literature: data sets on which the experiments are performed are most often not made available; the code is also usually not distributed, and even when it is, its usability is limited (e.g., because only the executable is provided and/or because the documentation is extremely terse or completely absent). Moreover, papers often do not describe the code in full detail, making a reimplementation almost impossible or useless. Finally, a comparison of time performances is made even more difficult by the relative impossibility of comparing structures offering different forms of access to the graph, and by the fact that they are implemented on hardware with different levels of computational power.

a run-length encoding to store such sequences (a technique similar to interval compression of [Boldi and Vigna 04]). With their machinery they achieve about 9.7 bits/link, with a random access time of about 3.5 μs/link (on a Solaris UltraSPARC-II with a 360-MHz CPU).

An independent and orthogonal approach to web-graph compression (or, more generally, to graph compression in general) tries to consider the problem from a structural viewpoint. In its simplest form, this solution may consist in classifying the arcs of the graphs into two or more classes and compressing them differently; an early example of this approach is attempted in [Suel and Yuan 01], which distinguishes between global frequent links (interhost links toward pages with large in-degree that are Huffman-coded), global absolute links (the other interhost links, stored using a Golomb code), and local links (which are, in turn, classified into "frequent" and "distant" links). Their approach requires about 13.9 bits/link with a random access time of about 0.26 ms/link (they do not give details about the hardware on which their experiments were run, though). A similar technique was discussed in [Raghavan and Garcia-Molina 03], where the nodes are first partitioned into groups (called supernodes), and both the quotient graph thereby obtained and each component graph are stored separately: with their approach, they occupy about 5.1 bits/link, and with sequential and random access require about 300 and 700 ns/link, respectively (on a 933-MHz Pentium III).

2.1. WebGraph

In this paper we refer to the BV compression scheme provided by the WebGraph framework [Boldi and Vigna 04]. It is a compression scheme that combines several ideas from the literature and some new insights in a carefully engineered algorithm:

- Successor lists are compressed by referencing previous successor lists—if this is advantageous—and the list of arcs copied by a previous successor list is represented by inclusion–exclusion blocks;
- consecutive successors are represented by intervals;
- the residual successors are gap-encoded using ζ codes [Boldi and Vigna 05], a kind of instantaneous code devised explicitly for power laws with small exponent.

WebGraph provides an implementation of the BV compression scheme in Java (full source is available). Moreover, the Laboratory for Web Algorithmics (LAW)[8]

[8] Available at http://law.dsi.unimi.it/.

provides freely several data sets in BV format: in what follows, when we name a graph, we will use its name in the LAW repository.

Clearly, the compression rate is strongly dependent on the ordering of the nodes: actually, on a randomly permuted graph, BV simply encodes the successor lists by gaps. We also remark that WebGraph is an extremely configurable system, in which, for instance, it is possible to choose a different instantaneous code for each component of the compression algorithm. By hard-wiring our currently best choices we could gain some constant speedup factor.

Another important design choice of WebGraph, which imposes a constant-time slowdown, is its *lazy* implementation. When the successor list of a node is computed, a small amount of memory is allocated to keep track of inclusion–exclusion blocks and intervals: the successors originated by the various aspects of the compression algorithm are represented by iterators, which are aggregated in a tree; at each node, the results are merged on the fly. As a result, random access does not require the allocation of memory for the entire list of successors of a node.

2.2. Recent Advances

In this section we gather more recent papers that have presented significant advances in compressed data structures for graphs. Most of the papers provide comparison results against WebGraph, which is considered a de facto standard in web-graph compression, and use data sets from the LAW repository.

An important remark is to be made about *pointers*: most, if not all, the structures we discuss provide both sequential and random access. The bits per link reported are those related to the data required for sequential access. For random access, however, some additional data must be loaded into memory: usually, a list of pointers into the bitstream representing the graph. This list is monotone, so it can be represented using Elias–Fano coding [Elias 74, Fano 71], requiring around $2 + \log \ell$ bits per pointer, where ℓ is the average number of bits per node. Since the overall space usage is of lower order with respect to the rest of the data (in practice, it is usually less than 10 bits/node), and it is a "common evil," it is often not reported in the literature. This lack of reporting creates some confusion, since it happens that in benchmark variants that do not compress, the pointers are used to achieve maximum speed: at that point, however, the impact of pointers on memory is very significant (i.e., it can cause double space usage).

Separable graphs. The work [Blandford et al. 03] has proposed a general-purpose compressed data structure that works for *separable graphs* (intuitively, a graph is separable if it can be split into small, approximately equal-sized disconnected pieces by removing a small number of edges). The representation does not seem to depend on the initial ordering; the compression rate and access time are good, but experiments are provided only for very small data sets, and the software is not available.

Pattern compression. The work [Asano et al. 08] uses different techniques to encode intra- and interhost links, and for intrahost compression they adopt six different kinds of patterns that are used to cover the local adjacency matrix. No code is made available, and experiments are performed just on small graphs. The random access time is on the order of several μs/link, so by our standards it is not classifiable as a compressed data structure.[9]

Compression by community detection. In [Buehrer and Chellapilla 08], the authors propose identifying *communities* (i.e., complete bipartite graphs) and replacing them with virtual nodes. They provide a scanning algorithm that can identify a significant number of such communities in a web graph, so that the final number of bits per link is better than WebGraph's (although in most cases it is worse than what we show in Table 3); experiments are performed on large graphs, too. No detailed data are provided on decompression time, however, albeit it is argued in [Karande et al. 09] that many useful algorithms can be run directly on virtual nodes. Since the software is not available, we have not been able to determine whether the compression scheme is dependent on the initial ordering, or to ascertain whether their solution can be classified as a compressed data structure.

Grammar compression. [Claude and Navarro 07] proposes a very interesting compression scheme based on a general-purpose grammar-based compression method, *Re-Pair* [Larsson and Moffat 99]. Re-Pair codes the most frequent pair of symbols with a new symbol and records compactly the corresponding expansion rule. The resulting scheme sometimes provides a better space/time improvement over WebGraph, and it is general-purpose, but is not as scalable because of the large amount of memory required and the long time required to compress a graph. Also in this case the authors claim that the method is coordinate-free, but experiments show that it is dependent on the initial ordering (on a random permutation of the LAW data set `uk-2002` we obtained 7.61 instead of 4.22 bits/link,

[9] Experiments compare the sequential access speed with that provided by WebGraph, but clearly the authors have mistakenly used the direct-access interface of WebGraph to perform a sequential scan, instead of employing an iterator (which scans the graph sequentially). Consequently, for high compression they show unrealistic data.

and compression required days rather than hours).[10] On LiveJournal, Re-Pair compression provides an unsatisfactory 18.37 bits/link.

k^2-trees. In [Brisaboa et al. 09], the authors propose another approach based on representing the adjacency *matrix* of the graph using a k^2-ary tree that records at each level which children contain at least a one. The tree is represented compactly using the vast array of knowledge about succinct data structures gathered in the last decade. There is a very important feature of this approach: the same structure can be used both to obtain the successor list and to obtain the predecessor list (apparently, also direct access would be easy to implement). The structure is about one order of magnitude slower than other approaches, but if both successors and predecessors are needed, the compression results are unbeatable. Of course, the initial ordering is essential (they use URL ordering), so also this approach is in principle amenable to improvements by permuting the underlying graph. Experiments are provided only for graphs up to 20 million nodes, and the software is not available. Since the method relies heavily on the 1's of the matrix being concentrated in relatively few locations, it is likely that this approach will have less success with social networks.

Compression by breadth-first visit. [Apostolico and Drovandi 09] proposes an innovative and very interesting method based on a breadth-first visit followed by a clever compression scheme that exploits naturally both locality and similarity: successors are compressed either by reference to the previous successor for the same node or by reference to the successor of the previous list that is in the same ordinal position. Moreover, blocks of identical successors are recorded just once. The authors provide an implementation of their algorithm (but no source code). Compression rates are excellent (in many cases, the best available), and access is very fast. The downside is that one is forced to store the graph *using a specific numbering of the nodes*. This problem can be obviated by storing the forward and reverse permutations and remapping each successor list on the fly, possibly sorting it. This approach, however, worsens significantly both space usage and access time. Thus, rather than with standard WebGraph, this technique is best compared with the compression rates of this paper, since the point of this paper is precisely choosing a permutation to maximize compression.[11]

[10] Since Re-Pair compression is not coordinate-free, it could in principle benefit from the permutations suggested in this paper. However, some preliminary experiments showed no improvement or even a worsening of the compression rate using the permutations described in this paper.

[11] Comparison with the Apostolico–Drovandi method did not appear in the preliminary version of this paper [Boldi et al. 09], since at that time, [Apostolico and Drovandi 09] had not yet been published.

The authors claim that the method is coordinate-free, and indeed, some experiments with random graphs show that the method has minimal dependence on the initial ordering (`uk-2002` requires 3.37 bits/link, whereas its random permutation requires 3.49), something that is unique among the methods we review. Preliminary experiments suggest that the proposed encoding method is strongly oriented toward graphs with tight clusters: the outcome on some social networks discussed in this paper is unsatisfactory (e.g., 15 bits/link on LiveJournal), and also results on transposed graphs are not very competitive (the best compression on the transpose of the data set `uk` of this paper is 1.348 bits/link).

Two observations must be made about this method. First of all, successors are not returned in increasing order. Second, to provide random access, additional data must be loaded into memory. These data include not only a set of pointers into the compressed data, but also a set of node identifiers. Thus, the "common evil" elimination that we discussed at the start of this section does not apply in this case. Moreover, in the current implementation, data are not kept in succinct form, a choice that provides access at a very high speed, but at the same time increases by a large amount the space allocated in main memory. To keep differences to a minimum, we compare results oriented to random access (e.g., Tables 4 and 5) to the "level-8" compression of the Apostolico–Drovandi scheme, which in the current implementation uses eight bits per node—a value comparable to that used by WebGraph for pointers alone.

3. Notation and Gray Code Basics

3.1. Von Neumann's Notation

In the following we will use von Neumann's definition of natural numbers

$$x = \{\, 0, 1, \ldots, x-1 \,\},$$

which leads to a simple notation for sets of integers. We allow some ambiguity when writing exponentials: 2^n denotes the vectors of n bits, or equivalently, the power set of n (interpreting the vectors as characteristic functions $n \to 2$). Since 2^n ambiguously also denotes the set $X = \{\, 0, 1, \ldots, 2^n - 1 \,\}$, we assume the natural correspondence between the latter set in increasing order and the strings of n bits in lexicographic ordering. The mapping from strings to X is obviously given by base-2 evaluation.

In the following, if $x \in 2^n$, we use $x_0, x_1, \ldots$ to denote the bits of its binary expansion (x_0 being its least-significant bit). In other words, interpreting x as a characteristic function $n \to 2$, we let

$$x_k = x(k).$$

x	$\bar{x}$
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Table 1. The natural Gray code on three bits: in the right-hand column, any two successive vectors differ in exactly one bit.

3.2. Gray Codes and Gray Orderings

An n-bit *Gray code* is an arrangement (i.e., a total ordering) of 2^n such that any two successive vectors differ by exactly one bit.[12] Gray codes, named after the physicist Frank Gray, find countless applications in computer science, physics, and mathematics (we refer the interested reader to [Knuth 05] for more information on this topic). The *ordering* imposed by a Gray code on 2^n is called a *Gray ordering*.

Even though there are many Gray codes, and thus many Gray orderings, one that is very simple to describe is the following.

Definition 3.1. [Knuth 05, Section 7.2.1.1] For every $x \in 2^n$, let $\bar{x} \in 2^n$ be defined by

$$\begin{aligned} \bar{x}_{n-1} &= x_{n-1}, \\ \bar{x}_k &= x_{k+1} \oplus x_k \quad \text{for } k < n-1, \end{aligned}$$

where $\oplus$ is the exclusive or. Then $x \mapsto \bar{x}$ is a bijection, and the ordering $\prec$ defined by

$$\bar{x} \prec \bar{y} \quad \text{iff} \quad x < y$$

is a Gray ordering, called the *natural Gray ordering*. (Table 1 shows the natural Gray code on three bits.)

Now let us write $x \mapsto \hat{x}$ for the inverse of $x \mapsto \bar{x}$; starting from Definition 3.1, one can easily see that $\hat{x}$ can be recursively obtained as follows:

$$\begin{aligned} \hat{x}_{n-1} &= x_{n-1}, \\ \hat{x}_k &= x_k \oplus \hat{x}_{k+1}. \end{aligned}$$

[12] We say that a' is the successor of a with respect to the ordering $<$ if $a < a'$ and $a \leq b \leq a'$ implies either $a = b$ or $b = a'$.

This observation gives a nice and simple way to compute $\hat{x}$ from x: indeed, let $x \downarrow k$ be the number of 1's in x preceding position k (i.e., the number of 1's in bits that are at least as significant as the kth). Then

$$\hat{x}_k = x \downarrow k \bmod 2. \tag{3.1}$$

4. On Gray Orderings and Graphs

An obvious application of Gray ordering to graphs is that of permuting node labels so that the resulting adjacency matrix changes "slowly" from row to row. Indeed, intuitively, if we permute the rows of the adjacency matrix following a Gray ordering, rows with a small number of changes should appear nearby. (This intuition is only partially justified; for example, in Table 1, the first and last words of the second column differ by just one bit, yet they are as far apart as possible). Of course, to maintain the graph correctly, we also need to permute columns in the same way, but this process will not change the number of differences between adjacent rows.

Indeed, some of the first investigations of web-graph compression experimented with Gray orderings [Randall et al. 02].[13] However, the authors reported a very small improvement in compression with respect to URL ordering; this fact, coupled with the obvious advantages of the latter, pushed the authors to discard Gray ordering altogether. The main question we try to answer in this paper is whether this small difference is actually an absolute property or an artifact strongly depending on the compression algorithms used, and whether it also applies to transposed graphs. To this purpose, we first develop a very simple algorithm that makes it possible to determine the Gray code ordering by inspecting the successor lists in parallel.

When manipulating web graphs using the WebGraph framework, successors are returned in the form of an iterator providing an increasing sequence of integers. This makes it possible to compare the position in the Gray ordering of two rows of the adjacency matrix by iterating in parallel over the adjacency lists of two nodes. While the lists coincide, we skip, and keep a variable recording the parity of the number of arcs seen so far (note that the value of (3.1) depends only on the *parity* of $x \downarrow k$). As soon as the lists differ, we can use formula (3.1) to compute the first different bit of the ranks of the adjacency rows in the Gray ordering: assuming that the first list returns j and the second list returns k (for the sake of simplicity, we assume that the end of the list is marked by ∞), we

[13] Note that in the paper the codes are incorrectly spelt as "Grey."

Algorithm 1. (Gray Ordering of Two Rows of Adjacency Matrix.)

The procedure for deciding the Gray natural order of two rows of the adjacency matrix, represented by means of iterators i and j that return the position of the next nonzero element. Note that the end of the list is denoted by ∞ and that we used Iverson's notation: $[a < b]$ has value one if $a < b$, zero otherwise. The return value is negative, null, or positive depending on whether iterator i corresponds to a vector that precedes, is equal to, or follows (respectively) the vector corresponding to iterator j according to the Gray natural order.

```
1. $p \leftarrow$ false;
2. forever begin
    (a) $a \leftarrow \mathrm{next}(i)$;
    (b) $b \leftarrow \mathrm{next}(j)$;
    (c) if $a = \infty$ and $b = \infty$ then return 0;
    (d) if $a \neq b$ then begin
        i. if $p \oplus [a < b]$ then return 1
        ii. else return $-1$
    (e) end;
    (f) $p \leftarrow \neg p$
3. end;
```

have the following scenario:

- if the parity is odd, the order of the lists is the order of j and k;
- if the parity is even, the order of the lists is the order of j and k reversed.

This can be easily seen because $j < k$ implies that the first difference in the rows of the adjacency matrix is at position j, where the first list has a one and the second list has a zero. If the parity is odd, this means that the rank of the first list has a zero in position j, while the rank of the second list has a one. The situation is reversed if the parity is even. Algorithm 1 describes this process formally.

Once this simple consideration is made, it is trivial to implement the Gray code (or lexicographic) graph permutation using WebGraph's facilities. The idea is to use a standard comparison-based sorting algorithm that compares lists

Name	Year	Nodes	Edges
cnr	2000	325,557	3,216,152
webbase	2001	118,142,155	1,019,903,190
it	2004	41,291,594	1,150,725,436
eu	2005	862,664	19,235,140
uk	2007	105,896,555	3,738,733,648

Table 2. Basic properties of graphs used as a data set.

of successors exploiting the considerations above.[14] As a result, we can compute the Gray permutation of the uk graph (see Table 2) in about one hour on an Opteron at 2.8 GHz.

Note that from a complexity viewpoint this approach is far from optimal. Indeed, a simple way to permute words in Gray code ordering is to apply a modified radix sort in which at each recursive call, we have a parity bit that tells us whether $0 \prec 1$ or $1 \prec 0$. We apply a standard radix sort algorithm, dividing words into two blocks depending on the first bit, and then recur on each block: however, when we recur on the block of words starting with one, we invert the parity bit.

This approach is theoretically optimal *if the size of the input is given by the number of entries in the adjacency matrix*. However, the adjacency matrix of a web graph is very sparse, and never represented explicitly.

Alternatively, we could develop a radix sort that picks up successors from successor lists (for all nodes) and deduces implicitly the zeros and the ones of the adjacency matrix. Albeit in principle such an algorithm would iterate optimally (i.e., it would extract from each iterator the minimum number of elements that are necessary to compute the ordering), it would require one to build at the same time the iterators for all nodes—a task that would require a preposterous amount of core memory.

5. Experimental Results

In this section, we present a number of experiments performed on web graphs and social graphs; the experiments show the impact of ordering on the compression performance of BV and provide positive evidence that Gray code ordering and its variants yield good compression ratios in all cases. As suggested in Section 2, we compare our results against the Apostolico–Drovandi compression method [Apostolico and Drovandi 09].

[14] In this case, the lazy architecture used by WebGraph turns out to be very effective, since the comparison ends very quickly for most pairs of successor lists, without the need to decompress them fully.

5.1. Experiments on Web Graphs

We ran a number of experiments on web graphs of different sizes (ranging from 300,000 nodes up to almost 120 million nodes) and collected at different times, and on their transposed versions. The graphs used are described in Table 2, and they are all publicly available, as well as the code used in the experiments.

We started with the standard URL ordering of nodes and permuted the nodes in different ways, taking note of the number of bits/link occupied if the graph is compressed in WebGraph format. Six node orderings were considered:

- *URL:* URL ordering (to avoid confusion, we use "URL ordering" instead of "URL lexicographic ordering");
- *lex:* lexicographic row ordering;
- *Gray:* Gray ordering;
- *LhbhGray:* loose host-by-host Gray ordering (i.e., keep URLs from the same host adjacent, and order them using Gray ordering);
- *ShbhGray:* strict host-by-host Gray ordering (as before, but Gray ordering is applied considering only local links, i.e., links to URLs of the same host).

We remark that we devised the latter two orderings in an attempt to combine external and internal information.

Figure 1 shows how the adjacency matrix changes when reordering is applied. To obtain this figure, we actually divided the matrix into smaller 100×100 square submatrices, computed the fraction of 1's found in each submatrix, and plotted a square using a gray scale that maps 0 to white and that becomes exponentially darker (1 is black).

WebGraph compression depends not only on the graph to be compressed and the ordering of its nodes, but also on a number of other parameters that control its behavior. Two parameters that happen to be important for our experiments are as follows:

- *Window size:* when compressing a certain matrix row, WebGraph compares it with a number of previous rows, and tries to compress it differentially with respect to (i.e., as a difference from) each such row, choosing at the end the row that gave the best compression (or none, if representing the row nondifferentially gives better compression); the number of rows considered in this process is called the *window size.* Of course, larger window sizes produce slower compression, but usually guarantee better compression.

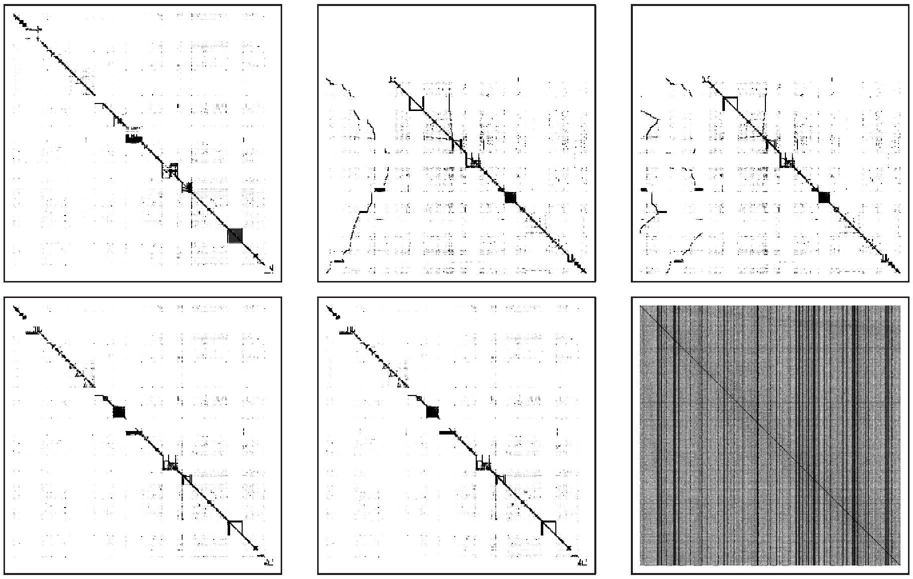

Figure 1. Adjacency matrix of the cnr graph (325,557 nodes) when nodes are ordered as follows (left to right, top to bottom): lexicographically by URL, lexicographically by row, Gray ordering, loose host-by-host Gray ordering, strict host-by-host Gray ordering, randomly.

- *Maximum reference count:* compressing a row x differentially with respect to some previous row $y < x$ makes it necessary, at access time, to decompress row y before decompressing row x; if also y is compressed differentially with respect to some other row $z < x$, z must be decompressed first, and so on, producing what we call a reference chain. This recursive process must be somehow limited, or access becomes extremely inefficient (and may even overflow the recursion stack). The *(maximum) reference count* is the maximum length of a reference chain (we simply avoid compressing a row differentially with respect to one that already has a maximum-length reference chain); the default reference count is 3, but this value may be pushed up to ∞ (meaning that we just don't care about creating too long chains: this makes sense if we plan to access the graph only sequentially, in which case we just need to keep the last w uncompressed rows, where w is the window size used for compression).

The Apostolico–Drovandi compression scheme, in contrast, needs a single parameter, the *level*, which determines the number of nodes stored in a compression block. Larger blocks imply better compression, but slower access (see [Apostolico and Drovandi 09] for details).

Graph	URL	lex	Gray	shbhGray	lhbhGray	Random	A&D
cnr	2.823	2.981	2.983	2.845	2.843	17.986	1.907
	2.654	2.185	2.192	2.176	2.177	15.084	2.055
webbase	3.059	3.410	3.416	2.907	2.895	30.937	2.883
	2.876	2.753	2.740	2.589	2.598	28.236	2.756
it	1.969	1.733	1.723	1.541	1.545	26.430	1.515
	1.737	1.206	1.207	1.206	1.209	21.717	1.412
eu	4.331	3.944	3.938	3.600	3.715	19.859	2.850
	3.903	2.832	2.833	2.761	2.795	16.445	2.543
uk	1.906	1.513	1.509	1.332	1.367	27.576	1.440
	1.662	1.042	1.040	1.007	1.014	21.682	1.348

Table 3. Compression rate summary; window size set to 8, and (maximum) reference count set to ∞. The level for the Apostolico–Drovandi method is set to 10,000. Every table cell contains the data for the graph and its transpose (A&D = [Apostolico and Drovandi 09]).

Graph	URL	lex	Gray	shbhGray	lhbhGray	Random	A&D
cnr	3.551	3.833	3.844	3.654	3.659	18.008	2.788
	2.839	2.489	2.495	2.472	2.474	15.084	2.408
webbase	3.732	4.404	4.409	3.680	3.688	30.937	3.770
	3.092	3.132	3.105	2.902	2.916	28.236	3.163
it	2.763	2.626	2.618	2.334	2.353	26.430	2.180
	1.852	1.407	1.393	1.378	1.379	21.717	1.586
eu	5.130	4.935	4.927	4.438	4.599	19.872	3.543
	4.002	3.063	3.072	2.993	3.019	16.445	2.902
uk	2.659	2.394	2.396	2.101	2.163	27.576	2.028
	1.761	1.222	1.205	1.162	1.170	21.682	1.493

Table 4. Compression rate summary; window size set to 8, and (maximum) reference count set to 3. The level for the Apostolico–Drovandi method is set to 8. Every table cell contains the data for the graph and its transpose (A&D = [Apostolico and Drovandi 09]).

Tables 3 and 4 show the numbers of bits/link occupied by the various graphs using ∞ and 3, respectively, as reference count (the window size was fixed at 8), and levels 10,000 and 8, respectively. Not surprisingly, the number of bits/link for the random permutation, which we present here only for comparison, is very large.

5.2. Experiments on Social Graphs

To see whether the results obtained on web graphs hold on when nonweb social networks are used, we took into consideration three data sets:

- DBLP[15] is a bibliography service from which a scientific collaboration network can be extracted: in this network each node represents a scientist,

[15] Available at http://www.informatik.uni-trier.de/~ley/db/.

and we interpret the coauthorship relation as a social tie, indicating that two scientists are connected if they have worked together on an article. We refer to this symmetric graph as the *DBLP social network*: as of now, it contains 326,186 scientists and 1,615,400 coauthorship relations. This particular social network has been studied in the past (e.g., see [Elmacioglu and Lee 05]), and it has been shown that it shares many properties of a typical social network, including the small-world property.

- One of the most popular social graphs is the graph of movie actors, also known as the *Hollywood graph*, in which two actors are joined by an edge if they appeared in a movie together. We used the Internet Movie Database[16] to build a symmetric graph with 1,139,905 nodes (actors and actresses) and 113,891,327 arcs.

- LiveJournal[17] is a virtual community social site begun in 1999; in this social network friendship is asymmetric—a user x can register y among his friends without asking y's permission. We considered the same 2008 snapshot of *LiveJournal* used in [Chierichetti et al. 09] for their experiments, consisting of 5,363,260 nodes and 100,060,170 directed friendship arcs.[18]

It is easy to reproduce on these graphs the intrinsic orderings used for web graphs presented in the previous section (lex, Random, Gray), but it is more difficult to provide the mixed orderings (lhbhGray and shbhGray) because there is no notion of "host" to refer to. Of course, one may use some external information (as in [Chierichetti et al. 09], where for LiveJournal the authors used the geographic information deduced from zip codes). Here we prefer to stick to intrinsic information only, and use a clustering algorithm to simulate the notion of host: more precisely, we fix a parameter k (the number of clusters) and use the graph-clustering tool METIS [Karypis et al. 98] to partition the graphs into k clusters, interpreting each cluster as a "host."

This approach introduces yet another parameter, and finding the optimal number of clusters becomes an issue that can be solved either a priori by finding the value k^* of k optimizing the structure of clusters themselves, or a posteriori by looking at the compression ratio produced at the end. Note that in principle, k^* may depend not only on the graph, but also on other compression parameters (e.g., the window size or the maximum reference count), and it may be different, depending on whether we are considering lhbh or shbh. In our experiments, though, there is no apparent dependence on other data, and k^* seems to depend

[16] Available at http://www.imdb.com/.

[17] Available at http://www.livejournal.com/.

[18] The data set was kindly provided by the authors of [Chierichetti et al. 09].

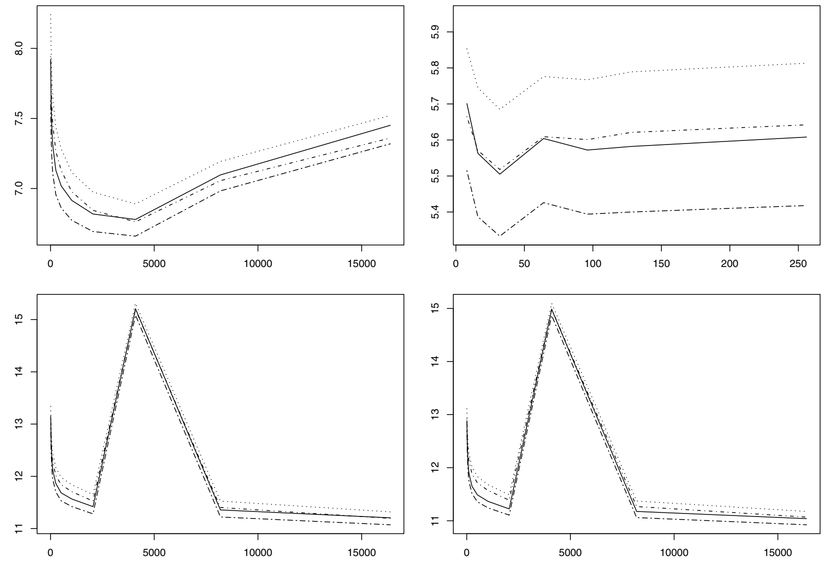

Figure 2. For each of the social graphs under consideration (left to right, top to bottom: DBLP, Hollywood, LiveJournal, and transposed LiveJournal), we show the number of bits/link as a function of k (the number of clusters), for loose/strict host-by-host Gray ordering and with reference count set to 3 or ∞ (window size was set to 8).

only on the graph: Figure 2 shows the compression ratio (bits/link) for the social graphs as a function of k, for the cases of both lhbh and shbh, and with window size set to 8 and reference count set to 3 or ∞. We considered various values of k ranging from $k = 8$ to $k = 16384$. As the reader can see, independently of all other parameters, the optimal k among those that we tested is $k^* = 4096$ for DBLP, $k^* = 32$ for the Hollywood graph, and $k^* = 16{,}384$ or even larger for LiveJournal and its transpose.

Graphs	**lex**	**Gray**	**shbhGray**	**lhbhGray**	**Random**	**A&D**
`dblp`	8.480	8.536	6.779	6.890	22.256	7.393
`hollywood`	6.293	6.291	5.505	5.685	16.247	7.644
`livejournal`	14.414	14.333	11.206	11.319	23.568	15.036
`livejournal (tr.)`	14.055	13.976	11.040	11.176	23.402	14.717

Table 5. Compression rate summary; window size set to 8, and (maximum) reference count set to 3. The level for the Apostolico–Drovandi method is set to 8 (A&D = [Apostolico and Drovandi 09]).

Graphs	lex	Gray	shbhGray	lhbhGray	Random	A&D
dblp	8.097	8.131	6.660	6.762	22.256	6.723
hollywood	6.045	6.062	5.333	5.518	16.247	7.478
livejournal	14.143	14.055	11.072	11.195	23.568	14.843
livejournal (tr.)	13.836	13.749	10.924	11.069	23.402	14.672

Table 6. Compression rate summary; window size set to 8, and (maximum) reference count set to ∞. The level for the Apostolico–Drovandi method is set to 10,000 (A&D = [Apostolico and Drovandi 09]).

Tables 5 and 6 present the compression ratio for the three data sets according to the different intrinsic orderings considered, using the optimal values k^* described above for lhbh and shbh.

5.3. Discussion

Our experiments highlight a number of very interesting points:

- *The effectiveness of intrinsic methods depends on the graph.* As we discussed, the fact that lex or Gray ordering should improve compression is intuitive, but has little formal justification. Indeed, on some graphs we have a visible decrease in compression.

- *Intrinsic methods are extremely effective on transposed graphs.* Our data for standard web graphs confirm what has been previously reported [Randall et al. 02], but our new data for transposed graphs show that here the situation is reversed: intrinsic methods improve very significantly the compression of transposed web graphs. With infinite reference chains, the transpose of `uk` requires just *one bit per link*. This is a phenomenon that clearly needs a combinatorial explanation.

  We modified WebGraph so that we could get access to detailed statistics about which compression technique is responsible for the increase in compression. Moving from URL to Gray ordering, the number of *copied* arcs of the `uk` graph (arcs that are not recorded explicitly, but rather represented differentially) jumps from $\approx 2\,\mathrm{G}$ to $\approx 3\,\mathrm{G}$. Thus, more than 80% of the arcs of the graph are not represented explicitly. Another visible effect is that of shifting the distribution of *gaps*—very small values (say below 10) are much more frequent, which also increases compression.

  We believe that such a major improvement in the transpose depends on the repetition in patterns of predecessors being much more frequent than in patterns of successors. For instance, all pages of level k are often pointed to by pages of level $k + 1$ (e.g., general topics of a site). This makes the predecessor list of level-k pages large and very similar. The same phenomenon does not happen for successors, because usually at level k the pointers at level $k+1$ are distinct. Moreover, URL ordering does not gather level-k pages nearby. Rather, it sorts URLs so that a level-k page is followed by all its subpages. In contrast, manual inspection of the URL permutation induced by strict host-by-host Gray ordering shows that pages on the same level of the hierarchy are grouped together.

- *Mixed methods work better.* Essentially in all cases, our new orderings outperform old methods.

- *The Apostolico–Drovandi method [Apostolico and Drovandi 09] is extremely effective, but not on transposed graphs and social networks.* The compression rates on large graphs are slightly better than those obtained using our best permutation (except for `uk` at high compression), but significantly worse on most transposed graphs. In particular, on the transpose of `uk` (our largest data set) we reach 1 bit/link, whereas the Apostolico–Drovandi method gives 1.348 bits/link. The results on social networks are always much worse than our best permutation, a phenomenon suggesting that their method is oriented toward networks with tight, almost disconnected clusters (i.e., web sites).

- *Social graphs exhibit differing compressions but similar behavior.* Our experiments on social graphs show that they react to node ordering much in the same way as web graphs, although the resulting compression ratio is worse and anyway extremely dependent on the graph considered, as already noted in [Chierichetti et al. 09]. It is interesting to note that our best result on LiveJournal is 11.072 bits/link, that is, only 16% more than the 9.559 bits/link obtained in [Chierichetti et al. 09] for the same graph using the more sophisticated compression scheme BL, which, however, is just a compression scheme, and not a data structure.

6. Conclusions

We have presented some experiments about the effect of permutations on the compression of web graphs and social networks. While our results are clearly preliminary, they highlight a number of issues that have not been tackled in the literature. First of all, we have provided two new permutations that significantly and uniformly increase WebGraph's compression rates. Second, we have shown that transposed graphs behave in a radically different manner when permuted with our techniques, giving rise to extreme compression rates. Third, we have shown that our methods are also effective for compressing social networks, although the outcome is less dramatic, which hints at significant structural differences.

Acknowledgments. This work was partially supported by the EC Project DELIS, by MIUR PRIN Project "Automi e linguaggi formali: aspetti matematici e applicativi," and by MIUR PRIN Project "Web Ram: web retrieval and mining." A preliminary version of the results in this paper appeared in [Boldi et al. 09].

References

[Adler and Mitzenmacher 01] Micah Adler and Michael Mitzenmacher. "Towards Compressing Web Graphs." In *DCC '01: Proceedings of the Data Compression Conference*, pp. 203–212. Washington, DC: IEEE Computer Society, 2001.

[Apostolico and Drovandi 09] Alberto Apostolico and Guido Drovandi. "Graph Compression by BFS." *Algorithms* 2:3 (2009), 1031–1044.

[Asano et al. 03] Yasuhito Asano, Tsuyoshi Ito, Hiroshi Imai, Masashi Toyoda, and Masaru Kitsuregawa. "Compact Encoding of the Web Graph Exploiting Various Power Laws (Statistical Reason behind Link Database)." In *Advances in Web-Age Information Management: 4th International Conference, WAIM 2003, Chengdu, China, August 17–19, 2003, Proceedings*, Lecture Notes in Computer Science 2762, pp. 37–46. New York: Springer, 2003.

[Asano et al. 08] Yasuhito Asano, Yuya Miyawaki, and Takao Nishizeki. "Effcient Compression of Web Graphs." In *Computing and Combinatorics: 14th Annual International Conference, COCOON 2008, Dalian, China, June 27–29, 2008, Proceedings*, Lecture Notes in Computer Science 5092, pp. 1–11. New York: Springer, 2008.

[Bharat et al. 98] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. "The Connectivity Server: Fast Access to Linkage Information on the Web." *Computer Networks and ISDN Systems* 30:1–7 (1998), 469–477.

[Blanco and Barreiro 05] Roi Blanco and Alvaro Barreiro. "Document Identifier Reassignment through Dimensionality Reduction." In *Advances in Information Retrieval: 27th European Conference on IR Research, ECIR 2005, Santiago de Compostela, Spain, March 21–23, 2005, Proceedings*, edited by David E. Losada and Juan M. Fernández-Luna, Lecture Notes in Computer Science 3408, pp. 375–387. New York: Springer, 2005.

[Blandford and Blelloch 02] Daniel K. Blandford and Guy E. Blelloch. "Index Compression through Document Reordering." In *Proceedings of the Data Compression Conference 2002*, pp. 342–351. Washington, DC: IEEE Computer Society, 2002.

[Blandford et al. 03] 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*, pp. 579–688. New York: ACM Press, 2003.

[Boldi and Vigna 04] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques." In *Proceedings of the Thirteenth International World Wide Web Conference*, pp. 595–601. New York: ACM Press, 2004.

[Boldi and Vigna 05] Paolo Boldi and Sebastiano Vigna. "Codes for the World Wide Web." *Internet Mathematics* 2:4 (2005), 407–429.

[Boldi et al. 09] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. "Permuting Web Graphs." In *Algorithms and Models for the Web-Graph: 6th International Workshop, WAW 2009, Barcelona, Spain, February 12–13, 2009, Proceedings*, Lecture Notes in Computer Science 5427, pp. 116–126. New York: Springer, 2009.

[Brisaboa et al. 09] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. "k^2-Trees for Compact Web Graph Representation." In *String Processing and Information Retrieval: 16th International Symposium, SPIRE 2009, Saariselk, Finland, August 25–27, 2009, Proceedings*, Lecture Notes in Computer Science 5721, pp. 18–30, New York: Springer, 2009.

[Buehrer and Chellapilla 08] Gregory Buehrer and Kumar Chellapilla. "A Scalable Pattern Mining Approach to Web Graph Compression with Communities." In *WSDM '08: Proceedings of the International Conference on Web Search and Web Data Mining*, pp. 95–106, New York: ACM, 2008.

[Chierichetti et al. 09] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. "On Compressing Social Networks." In *KDD '09: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 219–228, New York: ACM, 2009.

[Claude and Navarro 07] Francisco Claude and Gonzalo Navarro. "A Fast and Compact Web Graph Representation." In *String Processing and Information Retrieval: 14th International Symposium, SPIRE 2007, Santiago, Chile, October 29–31, 2007, Proceedings*, Lecture Notes in Computer Science 4726, pp. 118–129. New York: Springer, 2007.

[Elias 74] Peter Elias. "Effcient Storage and Retrieval by Content and Address of Static Files." *J. Assoc. Comput. Mach.* 21:2 (1974), 246–260.

[Elmacioglu and Lee 05] Ergin Elmacioglu and Dongwon Lee. "On Six Degrees of Separation in DBLP-DB and More." *SIGMOD Rec.* 34:2 (2005), 33–40.

[Fano 71] Robert M. Fano. "On the Number of Bits Required to Implement an Associative Memory." Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, MA, 1971.

[Guillaume et al. 02] Jean-Loup Guillaume, Matthieu Latapy, and Laurent Viennot. "Efficient and Simple Encodings for the Web Graph." In *Advances in Web-Age Information Management: WAIM '02: Third International Conference, WAIM 2002, Beijing, China, August 11-13, 2002, Proceedings*, pp. 328–337. New York: Springer, 2002.

[Jacobson 89] Guy Jacobson. "Space-Effcient Static Trees and Graphs." In *30th Annual Symposium on Foundations of Computer Science*, pp. 549–554. Research Triangle Park, NC: IEEE, 1989.

[Karande et al. 09] Chinmay Karande, Kumar Chellapilla, and Reid Andersen. "Speeding Up Algorithms on Compressed Web Graphs." In *WSDM '09: Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pp. 272–281. New York: ACM, 2009.

[Karypis et al. 98] George Karypis and Vipin Kumar. "Multilevel k-Way Partitioning Scheme for Irregular Graphs." *Journal of Parallel and Distributed Computing* 48 (1998), 96–129.

[Knuth 05] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*. Reading, MA: Addison-Wesley Professional, 2005.

[Kumar et al. 00] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tompkins, and Eli Upfal. "The Web as a Graph." In *PODS '00: Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 1–10. New York: ACM Press, 2000.

[Larsson and Moffat 99] N. Jesper Larsson and Alistair Moffat. "Offline Dictionary-Based Compression." In *DCC '99: Proceedings of the Conference on Data Compression*, pp. 296–305. Washington, DC: IEEE, 1999.

[Naor 90] M. Naor. "Succinct Representation of General Unlabeled Graphs." *Discrete Applied Mathematics* 28:3 (1990), 303–307.

[Raghavan and Garcia-Molina 03] Sriram Raghavan and Hector Garcia-Molina. "Representing Web Graphs." In *Proceedings of the 19th International Conference on Data Engineering, 2003*, pp. 405–416. Washington, DC: IEEE, 2003.

[Randall et al. 02] Keith H. Randall, Raymie Stata, Janet L. Wiener, and Rajiv G. Wickremesinghe. "The Link Database: Fast Access to Graphs of the Web." In *Proceedings of the Data Compression Conference 2002*, pp. 122–131. Washington, DC: IEEE, 2002.

[Shieh et al. 03] Wann-Yun Shieh, Tien-Fu Chen, Jean Jyh-Jiun Shann, and Chung-Ping Chung. "Inverted File Compression through Document Identifier Reassignment." *Inf. Process. Management* 39:1 (2003), 117–131.

[Silvestri 07] Fabrizio Silvestri. "Sorting Out the Document Identifier Assignment Problem." In *Advances in Information Retrieval: 29th European Conference on IR Research, ECIR 2007, Rome, Italy, April 2–5, 2007, Proceedings*, edited by Giambattista Amati, Claudio Carpineto, and Giovanni Romano, Lecture Notes in Computer Science 4425, pp. 101–112. New York: Springer, 2007.

[Suel and Yuan 01] Torsten Suel and Jun Yuan. "Compressing the Graph Structure of the Web." In *Proceedings of the Data Compression Conference 2001*, pp. 213–222. Washington, DC: IEEE, 2001.

[Turán 84] György Turán. "On the Succinct Representation of Graphs." *Discrete Applied Mathematics* 8:3 (1984), 289–294.

Paolo Boldi, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy (boldi@dsi.unimi.it)

Massimo Santini, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy (santini@dsi.unimi.it)

Sebastiano Vigna, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy (vigna@dsi.unimi.it)

Received June 30, 2009; accepted June 3, 2010.