Tight and Simple Web Graph Compression

Szymon Grabowski and Wojciech Bieniecki

Computer Engineering Department, Technical University of Łódź,
Al. Politechniki 11, 90–924 Łódź, Poland
{sgrabow,wbieniec}@kis.p.lodz.pl

Abstract. Analysing Web graphs has applications in determining page ranks, fighting Web spam, detecting communities and mirror sites, and more. This study is however hampered by the necessity of storing a major part of huge graphs in the external memory, which prevents efficient random access to edge (hyperlink) lists. A number of algorithms involving compression techniques have thus been presented, to represent Web graphs succinctly but also providing random access. Those techniques are usually based on differential encodings of the adjacency lists, finding repeating nodes or node regions in the successive lists, more general grammar-based transformations or 2-dimensional representations of the binary matrix of the graph. In this paper we present a Web graph compression algorithm which can be seen as engineering of the Boldi and Vigna (2004) method. We extend the notion of similarity between link lists, and use a more compact encoding of residuals. The algorithm works on blocks of varying size (in the number of input lines) and sacrifices access time for better compression ratio, achieving more succinct graph representation than other algorithms reported in the literature. Additionally, we show a simple idea for 2-dimensional graph representation which also achieves state-of-the-art compression ratio.

Keywords: graph compression, random access

1 Introduction

Development of succinct data structures is one of the most active research areas in algorithmics in the last years. A succinct data structure shares the interface with its classic (non-succinct) counterpart, but is represented in much smaller space, via data compression. Successful examples along these lines include text indexes [17], dictionaries, trees [12,16] and graphs [16]. Queries to succinct data structures are usually slower (in practice, although not always in complexity terms) than using non-compressed structures, hence the main motivation in using them is to allow to deal with huge datasets in the main memory. For example, indexed exact pattern matching in DNA would be limited to sequences shorter than 1 billion nucleotides on a commodity PC with 4 GB of main memory, if the indexing structure were the classic suffix array (SA), and even less than half of it, if SA were replaced with a suffix tree. On the other hand, switching to some compressed full-text index (see [17] for a survey) shifts the limit to over 10 billion nucleotides, which is more than enough to handle the whole human genome.

Another huge object of significant interest seems to be the Web graph. This is a directed unlabeled graph of connections between Web pages (i.e., documents), where the nodes are individual HTML documents and the edges from a given node are the outgoing links to other nodes. We assume that the order of hyperlinks in a document is irrelevant. Web graph analyses can be used to rank pages, fight Web spam, detect communities and mirror sites, etc. [11,20].

It was estimated that the graph of the Web index by *Yahoo!*, *Google*, *Bing* and *Ask* has between 21 and 59 billion nodes (`http://www.worldwidewebsize.com/`, May

Szymon Grabowski, Wojciech Bieniecki: Tight and Simple Web Graph Compression, pp. 127–137 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

2010), but the top figure is more likely. Therefore assuming 50 billion nodes and 20 outgoing links per node, we have about 1 trillion links. Using plain adjacency lists, representation of this graph would require about 8 TB, if the edges are represented with 64-bit pointers (note that 32-bit pointers may simply be too small). In a slightly less naïve variant, with 5-byte pointers (note that 40 bits are just enough to represent 1 trillion values, but cannot scale any longer), the space occupancy drops to 5 TB, i.e., is still ways beyond the capacities of the current RAM memories. We believe that, confronted with the given figures, the reader is now convinced about the necessity of compression techniques for Web graph representation.

2 Related work

We assume that a directed graph $G = (V, E)$ is a set of $n = |V|$ vertices and $m = |E|$ edges. The earliest works on graph compression were theoretical, and they usually dealt with specific graph classes. For example, it is known that planar graphs can be compressed into $O(n)$ bits [13,21]. For dense enough graphs, it is impossible to reach $o(m \log n)$ bits of space, i.e., go below the space complexity of the trivial adjacency list representation. Since the seminal Jacobson's thesis [14] on succinct data structures, there appear papers taking into account not only the space occupied by a graph, but also access times.

There are several works dedicated to Web graph compression. Bharat et al. [3] suggested to order documents according to their URL's, to exploit the simple observation that most outgoing links actually point to another document within the same Web site. Their Connectivity Server provided linkage information for all pages indexed by the AltaVista search engine at that time. The links are merely represented by the node numbers (integers) using the URL lexicographical order. We noted that we assume the order of hyperlinks in a document irrelevant (like most works on Web graph compression do), hence the link lists can be sorted, in ascending order. As the successive numbers tend to be close, differential encoding may be applied efficiently.

Randall et al. [19] also use this technique (stating that for their data 80% of all links are local), but they also note that commonly many pages within the same site share large parts of their adjacency lists. To exploit this phenomenon, a given list may be encoded with a reference to another list from its neighborhood (located earlier), plus a set of additions and deletions to/from the referenced list. Their encoding, in the most compact variant, encodes an outgoing link in 5.55 bits on average, a result reported over a Web crawl consisting of 61 million URL's and 1 billion links.

One of the most efficient compression schemes for Web graph was presented by Boldi and Vigna [4] in 2003. Their method is likely to achieve around 3 bits per edge, or less, at link access time below 1 ms at their 2.4 GHz Pentium4 machine. Of course, the compression ratios vary from dataset to dataset. We are going to describe the Boldi and Vigna algorithm in detail in the next section as this is the main inspiration for our solution.

Claude and Navarro [7,9] took a totally different approach of grammar-based compression. In particular, they focus on Re-Pair [15] and LZ78 compression schemes, getting close, and sometimes even below, the compression ratios of Boldi and Vigna, while achieving much faster access times. To mitigate one of the main disadvantages of Re-Pair, high memory requirements, they develop an approximate variant of this algorithm.

When compression is at a premium, one may acknowledge the work of Asano et al. [2] in which they present a scheme creating a compressed graph structure smaller by about 20–35% than the BV scheme with extreme parameters (best compression but also impractically slow). The Asano et al. scheme perceives the Web graph as a binary matrix (1s stand for edges) and detects 2-dimensional redundancies in it, via finding six types of blocks in the matrix: horizontal, vertical, diagonal, L-shaped, rectangular and singleton blocks. The algorithm compresses the data of intra-hosts separately for each host, and the boundaries between hosts must be taken from a separate source (usually, the list of all URL's in the graph), hence it cannot be justly compared to other algorithms mentioned here. Worse, retrieval times per adjacency list are much longer than for other schemes: on a order of a few milliseconds (and even over 28 ms for one of three tested datasets) on their Core2 Duo E6600 (2.40 GHz) machine running Java code. We note that 28 ms is at least twice more than the access time of modern hard disks, hence working with a naïve (uncompressed) external representation would be faster for that dataset (on the other hand, excessive disk use from very frequent random accesses to the graph can result in a premature disk failure). It seems that the retrieval times can be reduced (and made more stable across datasets) if the boundaries between hosts in the graph are set artificially, in more or less regular distances, but then also the compression ratio is likely to drop.

Also excellent compression results were achieved by Buehrer and Chellapilla [6], who used grammar-based compression. Namely, they replace groups of nodes appearing in several adjacency lists with a single "virtual node" and iterate this procedure; no access times were reported in that work, but according to findings in [8] they should be rather competitive and at least much shorter than of the algorithm from [2], with compression ratio worse only by a few percent.

Anh and Moffat [1] devised a scheme which seems to use grammar-based compression in a local manner. They work in groups of h consecutive lists and perform some operations to reduce their size (e.g., a sort of 2-dimensional RLE if a run of successive integers appears on all the h lists). What remains in the group is then encoded statistically. Their results are very promising: graph representations by about 15–30% (or even more in some variant) smaller than the BV algorithm with practical parameter choice (in particular, Anh and Moffat achieve 3.81 bpe and 3.55 bpe for the graph EU) and reported comparable decoding speed. Details of the algorithm cannot however be deduced from their 1-page conference poster.

Recent works focus on graph compression with support for bidirectional navigation. To this end, Brisaboa et al. [5] proposed the k^2-*tree*, a spatial data structure, related to the well-known quadtree, which performs a binary partition of the graph matrix and labels empty areas with 0s and non-empty areas with 1s. The non-empty areas are recursively split and labeled, until reaching the leaves (single nodes). An important component in their scheme is an auxiliary structure to compute *rank* queries [14] efficiently, to navigate between tree levels. It is easy to notice that this elegant data structure supports handling both forward and reverse neighbors, which implies from its symmetry. Experiments show that this approach uses significantly less space (3.3–5.3 bits per edge) than the Boldi and Vigna scheme applied for both direct and transposed graph, at the average neighbor retrieval times of 2–15 microseconds (Pentium4 3.0 GHz).

Even more recently, Claude and Navarro [8] showed how Re-Pair can be used to compress the graph binary relation efficiently, enabling also to extract the reverse

neighbors of any node. These ideas let them achieve a number of Pareto-optimal space-time tradeoffs, usually competitive to those from the k^2-tree.

3 The Boldi and Vigna scheme

Based on WebGraph datasets (`http://webgraph.dsi.unimi.it/`), Boldi and Vigna noticed that similarity is strongly concentrated; typically, either two adjacency (edge) lists have nothing or little in common, or they share large subsequences of edges. To exploit this redudancy, one bit per entry on the referenced list could be used, to denote which of its integers are copied to the current list, and which are not. Those bit-vectors are dubbed *copy lists*. Still, Boldi and Vigna go further, noticing that copy lists tend to contain runs of 0s and 1s, thus they compress them using a sort of run-length encoding. They assume the first run consists of 1s (if the copy list actually starts with 0s, the length of the first run is simply zero), and then it allows to represent a copy list as only a sequence of run lengths, encoded e.g. with Elias coding.

The integers on the current list which didn't occur on the referenced list must be stored too, and how to encode them is another novelty of the described algorithm. They detect intervals of consecutive (i.e., differing by 1) integers and encode them as pairs of the left boundary and the interval length; the left boundary of the next interval on a given list will be encoded as the difference to the right boundary of the previous interval minus two (this is because between the end of one interval and the beginning of another there must be at least one integer). The numbers which do not fall into any interval are called *residuals* and are also stored, encoded in a differential manner.

Finally, the algorithm allows to select as the reference list one of several previous lines; the size of the *window* is one of the parameters of the algorithm posing a tradeoff between compression ratio and compression/decompression time and space. Another parameter affecting the results is the maximum reference count, which is the maximum allowed length of a chain of lists such that one cannot be decoded without extracting its predecessor in the chain.

4 Our algorithm

Our algorithm (Alg. 1) works in blocks consisting of multiple adjacency lists. The blocks in their compact form are approximately equal, which means that the number of adjacency lists per block varies; for example, in graph areas with dominating short lists the number of lists per block is greater than elsewhere.

We work in two phases: preprocessing and final compression, using a general-purpose compression algorithm. The algorithm processes the adjacency lines one-by-one and splits their data into two streams.

One stream holds copy lists, in an extended sense compared to the Boldi and Vigna solution. Our copy lists are no longer binary but consist of four different flag symbols: 0 denotes an exact match (i.e., value j from the reference list occurs somewhere on the current list), 2 means that the current list contains integer $j + 1$, 3 means that the current list contains integer $j + 2$, if the corresponding integer from the reference list is j. Finally, the bits 1 correspond to the items from the reference list which have not been earlier labeled with 0, 2 or 3.

Alg. 1 GraphCompress($G, BSIZE$).

```
1   firstLine ← true
2   prev ← [ ]
3   outB ← [ ]
4   outF ← [ ]
5   for line ∈ G do
6       residuals ← line
7       if firstLine = false then
8           f[1 . . . |prev|] ← [1, 1, . . . , 1]
9           for i ← 1 to |prev| do
10              if prev[i] ∈ line then f[i] ← 0
11              else if prev[i] + 1 ∈ line then f[i] ← 2
12              else if prev[i] + 2 ∈ line then f[i] ← 3
13          append(outF, f)
14          for i ← 1 to |prev| do
15              if f[i] ≠ 1 then
16                  remove(residuals, prev[i])
17      residuals′ ← RLE(diffEncode(residuals)) + [0]
18      append(outB, byteEncode(residuals′))
19      prev ← line
20      firstLine ← false
21      if |outB| ≥ BSIZE then
22          compress(outB)
23          compress(outF)
24          outB ← [ ]
25          outF ← [ ]
26          firstLine ← true
```

Of course, several events may happen for a single element, e.g., the integer 34 from the reference list triggers three events if the current list contains 34, 35 and 36. In such case, the flag with the smallest value is chosen (i.e., 0 in our example).

Moreover, we make things even simpler than in the Boldi–Vigna scheme and our reference list is always the previous adjacency list.

The other stream stores residuals, i.e., the values which cannot be decoded with flags 0, 2 or 3 on the copy lists. First differential encoding is applied and then an RLE compressor for differences 1 only (with minimum run length set experimentally to 5) is run. The resulting sequence is terminated with a unique value (0) and then encoded using a byte code.

For this last step, we consider two variants. One is similar to *two-byte dense code* [18] in spending one bit flag in the first codeword byte to tell the length of the current codeword. Namely, we choose between 1 and b bytes for encoding each number, where b is the minimum integer such that $8b - 1$ bits are enough to encode any node value in a given graph. In practice it means that $b = 3$ for EU and $b = 4$ for the remaining available datasets.

The second coding variant can be classified as a prelude code [10] in which two bits in the first codeword byte tell the length of the current codeword; originally the lengths are 1, 2, 3 and 4 but we take 1, 2 and b such that $8b - 2$ bits are enough to encode the largest value in the given graph (i.e., b could be 5 or 6 for really huge graphs).

Once the residual buffer reaches at least BSIZE bytes, it is time to end the current block and start a new one. Both residual and flag buffers and then (independently) compressed (we used the well-known Deflate algorithm for this purpose) and flushed.

The code at Alg. 1 is slightly simplified; we omitted technical details serving for finding the list boundaries in all cases (e.g., empty lines).

5 Experimental results

We conducted experimented on the crawls EU-2005 and Indochina-2004, downloaded from the WebGraph project (`http://webgraph.dsi.unimi.it/`), using both direct and transposed graphs. The main characteristics of those datasets are presented in Table 1.

Dataset	EU-2005		Indochina-2004	
	direct	transposed	direct	transposed
Nodes	862664		7414866	
Edges	19235140		19235140	
Edges / nodes	22.30		26.18	
% of empty lists	8.31	0.000	17.66	0.004
Longest list length	6985	68922	6985	256425

Table 1. Selected characteristics of the datasets used in the experiments.

The main experiments (Sect. 5.1) were run on a machine equipped with an Intel Core 2 Quad Q9450 CPU, 8 GB of RAM, running Microsoft Windows XP (64-bit). Our algorithms were implemented in Java (JDK 6). A single CPU core was used by all implementations. As seemingly accepted in most reported works, we measure access time per edge, extracting many (100,000 in our case) randomly selected adjacency lists and summing those times, and dividing the total time by the number of edges on the required lists. The space is measured in bits per edge (bpe), dividing the total space of the structure (including entry points to blocks) by the total number of edges.

Throughout this section by 1 KB we mean 1000 bytes.

5.1 Compression ratios and access times

Our routine has three parameters: the number of flags used (either 2 or 4, where 2 flags mimic the Boldi–Vigna scheme and 4 correspond to Alg. 1), the byte encoding scheme (either using 2 or 3 codeword lengths), and the residual block size threshold BSIZE. As for the last parameter, we initially set it to 8192, which means that the residual block gets closed and is submitted to the Deflate compression once it reaches at least 8192 bytes. Experiments with the block size are presented in the next subsection. The remaining parameters constitute four variants:

2a Two flags and two codeword lengths are used.
2b Two flags and three codeword lengths are used.
4a Four flags and two codeword lengths are used.
4b Four flags and three codeword lengths are used.

Dataset	EU-2005		Indochina-2004	
	direct	transposed	direct	transposed
2a	2.286	2.345	1.101	1.087
2b	2.199	2.290	1.062	1.065
4a	1.735	1.809	0.936	0.903
4b	1.696	1.782	0.909	0.890

Table 2. Compression ratios in bits per edge.

As expected, the compression ratios improve with using more flags and more dense byte codes (Table 2). Tables 3 and 4 present the compression and access time results for the two extreme variants: 2a and 4b. Here we see that using more aggressive preprocessing is unfortunately slower (partly because of increased amount of flag data per block) and the difference in speed between variants 2a and 4b is close to 50%. Translating the times per edge into times per neighbor list, we need from 410 μs to 550 μs for 2a and from 620 μs to 760 μs for 4b. This is about 10 times less than the access time of 10K or 15K RPM hard disks.

	direct graph		transposed graph	
	bpe	time [μs]	bpe	time [μs]
BV (7,3)	5.169	0.24	–	–
2a	2.286	18.59	2.345	18.88
4b	1.696	28.93	1.782	27.83

Table 3. EU-2005 dataset. Compression ratios (bpe) and access times per edge. To the results of BV (7,3) the amount of 0.510 bpe should be added, corresponding to extra data required to access the graph in random order.

	direct graph		transposed graph	
	bpe	time [μs]	bpe	time [μs]
BV (7,3)	2.063	0.21	–	–
2a	1.101	20.77	1.087	21.10
4b	0.909	29.03	0.890	27.43

Table 4. Indochina-2004 dataset. Compression ratios (bpe) and access times per edge. To the results of BV (7,3) the amount of 0.348 bpe should be added, corresponding to extra data required to access the graph in random order.

5.2 Varying the block size

Obviously, the block size should seriously affect the overall space used by the structure and the access time. Larger blocks mean that the Deflate algorithm is more successful in finding longer matches and the overhead from encoding first lines in a block without any reference is smaller. On the other hand, more lines have to be usually decoded before extracting the queried adjacency list.

In this experiment we run the 2a algorithm (the same implementation in Java) with each block of residuals terminated (and later Deflate-compressed) after reaching BSIZE of 1024, 2048, 4096, 8192 and 16384 bytes, respectively. The test computer had an Intel Pentium4 HT 3.0 GHz CPU, 1 GB of RAM, and was running Microsoft Windows XP Home SP3 (32-bit). The results (Table 5) show that doubling the block size implies space reduction by about 10% while the access time grows less than twice (in particular, using 8K blocks is only 2.0–2.5 times slower than using 2K blocks). Still, as the block size gets larger (compare the last two rows in the table), the improvement in compression starts to drop while the slowdown grows. For a reference, the access times of a practical Boldi–Vigna variant, BV (7,3), are 0.47 μs and 0.42 μs on the test machine.

	EU-2005		Indochina-2004	
	bpe	time [μs]	bpe	time [μs]
1024	3.398	6.50	1.485	8.99
2048	2.869	8.91	1.292	12.05
4096	2.513	15.93	1.172	17.87
8192	2.286	27.60	1.101	29.83
16384	2.129	48.77	1.061	57.39

Table 5. Compression ratios and access times in function of the block size. 2a variant used. Tests run on the non-transposed graphs.

6 Obtaining forward and reverse neighbors

Sometimes one is interested in grasping not only the (forward) neighbors of a given node but also the nodes that point to the current node (also called its *reverse neighbors*). A naïve solution to this problem is to store a twin data structure built for the transposed graph, which more or less doubles the required space. Interestingly, as pointed out in Sect. 2, more sophisticated ideas are already known, using 2D structures that support bidirectional navigation over the graph.

In this section we propose two simple techniques for this problem scenario. One of them reduces the size of the compressed transposed graph for the price of moderate increase in search time. Basically, the idea is to remove parts of some adjacency lists from the transposed graph and refer to the compressed structure for the direct graph when there is a need to extract those removed reverse neighbors. In our preliminary experiments the transposed graph compressed component was reduced by less than 10% while for many lists the access time had to be approximately doubled (instead of extracting one compressed block, two randomly accessed blocks had to be extracted). Even if more can be done along these lines, we do not anticipate this approach being competitive.

The other algorithm partitions the binary matrix of the EU graph into squares, in the manner of the k^2-tree, but without any hierarchy, i.e., using only one level of blocks. Although seemingly very primitive, this idea let us attain the smallest space ever reported in the literature, for the EU dataset, among the algorithms supporting bidirectional navigation, namely 1.76 bpe, but the average extraction time per adjacency list is now on the order of a few milliseconds, i.e., close to hard disk access time. This is, in a way, an extreme result; a slower algorithm could already lose in speed to a plain external representation.

In an experiment, we partitioned the binary matrix M of the EU graph ($n = 862,664$ nodes) into boxes (squares) of size $B = 1024$ (the boundary areas may be rectangular). Each box is identified with a single bit (totalling 89 KB) where 1s stand for the non-empty boxes (those that contain at least one edge). The non-empty boxes, obtained in a row-wise scan, are labeled with successive integers, which are offsets in an array $A[1 \ldots |A|]$ of pointers to the actual (compressed) content of the corresponding boxes. Now we present how forward and reverse neighbors of a given page are found.

To find the forward neighbors of page j, we must retrieve and decode all the non-empty boxes overlapping the jth row of the matrix. Note that for efficient retrieval we need only to find quickly in the array A the pointer to first (leftmost) such box as all its successors will be pointed from the following cells of A. A trivial yet satisfying

solution is to store in an extra array the indexes in A of the leftmost non-empty boxes for the following rows of boxes. This needs n/B indexes (about 3.3 KB for the EU graph if 4-byte indexes are used).

Finding the reverse neighbors is harder but we avoid the challenge and solve it trivially, storing an array analogous to A, only built according to the column-wise scan. For the EU graph and our choice of B, the number of non-empty blocks is about 24,700, i.e., the extra cost in space is just above 100 KB (4-byte pointers and the 3.3 KB of the auxiliary array).

As mentioned, the non-empty boxes are stored in compressed form. We (conceptually) flatten each box, writing it row after row, and encode the gaps between the successive 1s. The gaps are represented with a byte code (1, 2 or 3 bytes per gap). Finally, the sequence of encoded gaps is compressed with the Deflate algorithm. To improve compression, for each non-empty box we check if transposing it results in a smaller Deflate-compressed size and also if it is better not to compress it at all (data expansion is typical if the box contains only a few items). This adds two extra bits per non-empty box.

Note that accessing the (forward or reverse) neighbors of a given page requires decoding many boxes, even those that have no item in common with the desired neighbor list. For the EU graph a single list passes through about 29 non-empty boxes, on average. The average non-empty box occupies a little over 1.3 KB before Deflate compression (their size variance is however very large), which means that retrieving the neighbor list requires extracting compressed data to about 39 KB, on average. This estimation is optimistic since decompressing a single chunk of data is usually faster than of several chunks totalling the same size, because of the locality of memory accesses. Moreover, as said, those average-case estimations are far from the worst case. Yet another factor is that the decoded boxes must be filtered to return only those values which belong to the desired list. Overall, we however believe that one can retrieve the neighbor list in about 2–3 ms (i.e., about 100 microseconds per neighbor) on modern hardware in an average case.

The total size of the EU graph compressed in the presented way is about 4,225 KB, which translates to 1.76 bpe. This contrasts with 3.93 bpe presented as the most succinct result in [8], for which graph representation the average reported direct and reverse edge retrieval time is about 35 and 55 microseconds, respectively. As the number of edges per adjacency list is about 22 for this graph, the times to extract the whole list are close to 1 ms which is not that far from our (very crudely estimated) retrieval times. This leads us to the conclusion that even simple heuristics and off-the-shelf tools (like the Deflate compression algorithm) may help one get close to the state of the art and should encourage researchers to rethink the problem.

7 Conclusions

We presented two algorithms for Web graph compression, one encoding blocks consisting of whole lines and the other working on boxes (squares) of the graph binary matrix. Both algorithms achieve much better compression results than those presented in the literature, although for the price of relatively slow access time. We point out, however, that one extreme tradeoff in succinct in-memory data structures is when accessing the structure is only slightly faster than reading data from disk. The niche for such a solution is when the given Web crawl cannot fit in RAM memory using less tight compressed representation and the stronger compression is already

enough. The disk transfer rate is of relatively small imporantance here and what matters is the access time, which is about 10 ms or more for commodity 7200 RPM hard disks. Our algorithms spend significantly less time for extracting an average adjacency list, even if they are 1 or 2 orders of magnitude slower than the solutions from [4,7,8]. Another challenge is to compete with SSD disks which are not much faster than conventional disks in reading or writing sequential data but their access times are two orders of magniture smaller.

Our future work will focus on improving the access times; some possibilities lie in more aggressive reference list encoding via referring to several (cf. [4]) rather than a single previous list, using smaller independently compacted blocks with backend compression applied over many of them, and replacing Deflate with alternative compressors from LZ77 family, e.g. LZMA (`http://www.7-zip.org/`).

8 Acknowledgments

The work was partially supported by the Polish Ministry of Science and Higher Education under the project N N516 477338 (2010–2011).

References

1. V. N. Anh and A. F. Moffat: *Local modeling for webgraph compression*, in DCC, J. A. Storer and M. W. Marcellin, eds., IEEE Computer Society, 2010, p. 519.
2. Y. Asano, Y. Miyawaki, and T. Nishizeki: *Efficient compression of web graphs*, in COCOON, X. Hu and J. Wang, eds., vol. 5092 of Lecture Notes in Computer Science, Springer, 2008, pp. 1–11.
3. K. Bharat, A. Z. Broder, M. R. Henzinger, P. Kumar, and S. Venkatasubramanian: *The Connectivity Server: Fast access to linkage information on the Web.* Computer Networks, 30(1–7) 1998, pp. 469–477.
4. P. Boldi and S. Vigna: *The webgraph framework I: Compression techniques*, in WWW, S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, eds., ACM, 2004, pp. 595–602.
5. N. Brisaboa, S. Ladra, and G. Navarro: *K2-trees for compact web graph representation*, in Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 5721, Springer, 2009, pp. 18–30.
6. G. Buehrer and K. Chellapilla: *A scalable pattern mining approach to web graph compression with communities*, in WSDM, M. Najork, A. Z. Broder, and S. Chakrabarti, eds., ACM, 2008, pp. 95–106.
7. F. Claude and G. Navarro: *Fast and compact Web graph representations*, Tech. Rep. TR/DCC-2008-3, Department of Computer Science, University of Chile, April 2008.
8. F. Claude and G. Navarro: *Extended compact web graph representations*, in Algorithms and Applications, T. Elomaa, H. Mannila, and P. Orponen, eds., vol. 6060 of Lecture Notes in Computer Science, Springer, 2010, pp. 77–91.
9. F. Claude and G. Navarro: *Fast and compact web graph representations.* ACM Transactions on the Web (TWEB), 2010, To appear.
10. J. S. Culpepper and A. Moffat: *Enhanced byte codes with restricted prefix properties*, in SPIRE, M. P. Consens and G. Navarro, eds., vol. 3772 of Lecture Notes in Computer Science, Springer, 2005, pp. 1–12.
11. D. Donato, L. Laura, S. Leonardi, U. Meyer, S. Millozzi, and J. F. Sibeyn: *Algorithms and experiments for the webgraph.* J. Graph Algorithms Appl., 10(2) 2006, pp. 219–236.
12. R. F. Geary, N. Rahman, R. Raman, and V. Raman: *A simple optimal representation for balanced parentheses*, in Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings, S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, eds., vol. 3109 of Lecture Notes in Computer Science, Springer–Verlag, 2004, pp. 159–172.

13. X. He, M.-Y. Kao, and H.-I. Lu: *A fast general methodology for information-theoretically optimal encodings of graphs.* SIAM J. Comput., 30(3) 2000, pp. 838–846.
14. G. Jacobson: *Succinct Static Data Structures*, PhD thesis, Carnegie Mellon University, 1989.
15. N. J. Larsson and A. Moffat: *Off-line dictionary-based compression.* Proceedings of the IEEE, 88(11) Nov. 2000, pp. 1722–1732.
16. J. I. Munro and V. Raman: *Succinct representation of balanced parentheses, static trees and planar graphs*, in IEEE Symposium on Foundations of Computer Science (FOCS), 1997, pp. 118–126.
17. G. Navarro and V. Mäkinen: *Compressed full-text indexes.* ACM Computing Surveys, 39(1) 2007, p. article 2.
18. P. Procházka and J. Holub: *New word-based adaptive dense compressors*, in IWOCA, J. Fiala, J. Kratochvíl, and M. Miller, eds., vol. 5874 of Lecture Notes in Computer Science, Springer, 2009, pp. 420–431.
19. K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener: *The link database: Fast access to graphs of the Web*, 2001.
20. H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara: *A large-scale study of link spam detection by graph algorithms*, in AIRWeb '07: Proceedings of the 3rd international workshop on Adversarial information retrieval on the web, New York, NY, USA, 2007, ACM, pp. 45–48.
21. G. Turán: *On the succinct representation of graphs.* Discrete Applied Math, 15(2) May 1984, pp. 604–618.