Discrete Applied Mathematics 163 (2014) 298–306

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Tight and simple Web graph compression for forward and reverse neighbor queries

Szymon Grabowski *, Wojciech Bieniecki

Technical University of Łódź, Computer Engineering Department, Al. Politechniki 11, 90–924 Łódź, Poland

ARTICLE INFO

Article history:
Received 21 December 2011
Received in revised form 24 April 2013
Accepted 27 May 2013
Available online 17 June 2013

Keywords:
Graph compression
Random access
Bidirectional neighbor queries

ABSTRACT

Analyzing 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 three Web graph compression algorithms. The first can be seen as engineering of the Boldi and Vigna (2004) [8] 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 lists) and sacrifices access time for better compression ratio, achieving more succinct graph representation than other algorithms reported in the literature. The second algorithm works on blocks of the same size in the number of input lists. Its key mechanism is merging the block into a single ordered list. This method achieves much more attractive space–time tradeoffs. Finally, we present an algorithm for bidirectional neighbor query support, which offers compression ratios better than those known from the literature.

1. Introduction

Development of succinct data structures (e.g. text indexes [25], dictionaries and trees [24,16]) 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. 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. An example could be human genome which can be easily represented in less than 2 GB of RAM with a compressed full-text index (see [25] for a survey).

Another huge object of significant interest is the Web graph. This is a directed unlabeled graph of connections between webpages (i.e. documents), where the nodes are individual HTML documents and the edges from a given node are the outgoing links to other documents. 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..

As of Dec. 2011, it is estimated that Google's index has about 52 billion webpages.[1] Assuming 20 outgoing links per node, 5-byte links (4-byte indexes to other pages are simply too small) and pointers to each adjacency list we would need more than 5.2 TB of memory, ways beyond the capacities of the current RAM memories.

* Corresponding author. Fax: +48 426312755.
E-mail address: sgrabow@kis.p.lodz.pl (S. Grabowski).

[1] http://www.worldwidewebsize.com/.

0166-218X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.dam.2013.05.028

Preliminary versions of this manuscript were published in [17,18].

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 [28,19]. 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 [21] 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. [4] 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. [27] 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 solution (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 influential works in the Web graph compression area was presented by Boldi and Vigna [8] in 2003. Their method is likely to achieve around 3–5 bits per edge at link access time below 1 μs at their system (2.4 GHz Pentium4, 512 MB RAM, Java implementation). They 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 redundancy, 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, called *copy lists*, tend to contain runs of 0s and 1s, thus they are compressed with a sort of run-length encoding with lengths represented compactly. The integers on the current list, which did not occur on the referenced list, must be stored too. In the Boldi and Vigna scheme intervals of consecutive (i.e. differing by 1) integers are detected and encoded as pairs of the left boundary and the interval length. The left boundary of the next interval will be encoded with reference to the previous interval. 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 one of several previous lines as the reference list; 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.

Claude and Navarro [12,13] took a totally different approach of grammar-based compression. In particular, they focus on Re-Pair [23] and LZ78 compression schemes getting ratios close and sometimes even below results obtained by Boldi and Vigna, while achieving much faster access times. To mitigate one of the main disadvantages of Re-Pair, high memory requirements, they developed an approximate variant of this algorithm.

When compression is at a premium, one may acknowledge the work of Asano et al. [3], 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: in the order of a few milliseconds (and even over 28 ms for one of three tested datasets) on their Core 2 Duo E6600 machine (2.40 GHz, 2 GB RAM) running Java code. We note that 28 ms is at least twice 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 [10], who used grammar-based compression. 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 [14] they should be rather competitive and at least much shorter than of the algorithm from [3] with a compression ratio worse only by a few percent.

Apostolico and Drovandi [2] proposed an alternative Web graph ordering, reflecting their BFS traversal (starting from a random node) rather than traditional URL-based order. They obtain quite impressive compressed graph structures, often by 20%–30% smaller than those from BV at comparable access speeds. Interestingly, the BFS ordering allows to handle the link existential query (testing if page i has a link to page j) almost two times faster than returning the whole neighbor list. Still,

we note that using non-lexicographical ordering is harmful for compact storing of the webpage URLs themselves (a problem accompanying pure graph structure compression in most practical applications). Note also that reordering the graph is the approach followed in more recent works from the Boldi and Vigna team [7,6].

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. Although their results were promising, 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. [9] 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 [21] 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. Ladra [22] proposed a more efficient encoding of leaves (which are boxes of sizes e.g. 8×8 rather than single bits) in this scheme, making use of a common vocabulary for the different leaf submatrices and directly addressable codes. On the base of the mentioned encoding Claude and Ladra [11] achieved even better results. Their key idea was to divide the original square matrix into subdomains cutting out several non-overlapping squares (subgraphs) along the diagonal of the binary matrix; each generated subgraph is stored independently. Experiments show that even the original work uses significantly less space (3.3–5.3 bits per link) than the Boldi and Vigna scheme applied for both direct and transposed graph at the average neighbor retrieval times of 2–15 ms (our tests were on a Pentium4 HT 3.0 GHz, 1 GB RAM, Windows XP, Java 6). The Claude and Ladra variant reduces the space to about 3–4 bits per link and the retrieval time is improved to about 1 ms or less (Intel Xeon 2.0 GHz, 8 cores, 16 GB RAM, Linux 2.6, C++ , gcc).

In another recent work, Claude and Navarro [14] 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 with those from the (original variant of the) k^2-tree.

Finally, we have to mention the Hernández and Navarro work [20], where they combine their previous techniques, k^2-tree [9] and Re-Pair for compressing the graph binary relation [14] with edge reducing [10], obtaining interesting trade-offs. In particular, if some of the access time can be sacrificed, the space they achieved is the smallest known among the solutions supporting bidirectional queries.

3. Our algorithms for forward neighbors only

In this section we present two algorithms handling forward neighbors only (to handle reverse neighbors, the same algorithm must be run on the transposed graph). They work locally, in small blocks. The first of them usually reaches slightly higher compression ratios, but the second is more practical, as it is much faster in extracting the neighbor lists.

3.1. An algorithm based on similarity of successive lists

Our first algorithm (Fig. 1, SSL stands for "similarity of successive lists") 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.

If value j from the reference list occurs somewhere on the current list, flag 0 is emitted. Otherwise, if the current list contains $j+1$, flag 2 is sent. Otherwise, if the current list contains $j+2$, flag 3 is sent. Finally, flag 1 corresponds to an item from the reference list which has not been earlier labeled with 0, 2 or 3. (Using even more such flags could be possible, but with significantly diminishing returns, hence we chose some compromise between compression ratio, compression/extraction speed and simplicity of the scheme.)

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 solution is similar to [26] in spending one bit flag in the first codeword byte to tell the length of the current codeword. 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 is $b=3$ for small and $b=4$ for large datasets. The second coding variant can be classified as a prelude code [15] in which two bits in the first codeword byte tell the length of the current codeword; originally the lengths are 1, 2, 3, 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).

```
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
```

Fig. 1. GraphCompressSSL(*G*, BSIZE).

```
1   outF ← [ ]
2   i ← 1
3   for line_i, line_{i+1}, . . . , line_{i+h−1} ∈ G do
4       tempLine_1 ← line_i ∪ line_{i+1} ∪ . . . ∪ line_{i+h−1}
5       tempLine_2 ← removeDuplicates(tempLine_1)
6       longLine ← sort(tempLine_2)
7       items ← diffEncode(longLine) + [0]
8       outB ← byteEncode(items)
9       for j ← 1 to |longLine| do
10          f[1 . . . |longLine|] ← [0, 0, . . . , 0]
11          for k ← 1 to h do
12              if longLine[j] ∈ line_{i+k−1} then f[k] ← 1
13          append(outF, bitPack(f))
14      compress(concat(outB, outF))
15      outF ← [ ]
16      i ← i + h
```

Fig. 2. GraphCompressLM(*G*, BSIZE).

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 are then (independently) compressed (we used the well-known Deflate algorithm for this purpose) and flushed.

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

3.2. An algorithm based on list merging

Our second algorithm (Fig. 2, LM stands for "list merging") works in blocks having the same number of lists, h (at least in this aspect our algorithm resembles the one from [1]).

Given the block of h lists, the procedure converts it into two streams: one stores one long list consisting of all integers on the h input lists, without duplicates, and the other stores flags necessary to reconstruct the original lists. In other words, the algorithm performs a reversible merge of all the lists in the block.

The long list is compacted in a manner similar to the previous algorithm: the list is differentially encoded, zero-terminated and submitted to a byte coder (the variant with 1, 2 and b bytes per codeword was only tried). Here we resign from RLE compression; similar redundancy may be exploited by Deflate in a later stage without introducing additional delays.

The flags describe to which input lists a given integer on the output list belongs; the number of bits per each item on the output list is h, and in practical terms we assume h being a multiple of 8 (and even additionally a power of 2, in the experiments to follow). The flag sequence does not need any terminator since its length is defined by the length of the long list which is located earlier in the output stream. For example, if the length of the long list is 91 and $h = 32$, the corresponding flag sequence has 364 bytes.

Now, we consider two variations for encoding the flag sequence: either they are kept raw (the variant is later denoted as *LM-bitmap*), or differences (gaps) between the successive 1s in the flag sequence are written on individual bytes (the variant is latter denoted as *LM-diff*). We note that each run of h bits corresponding to flags for a single value on the output list must

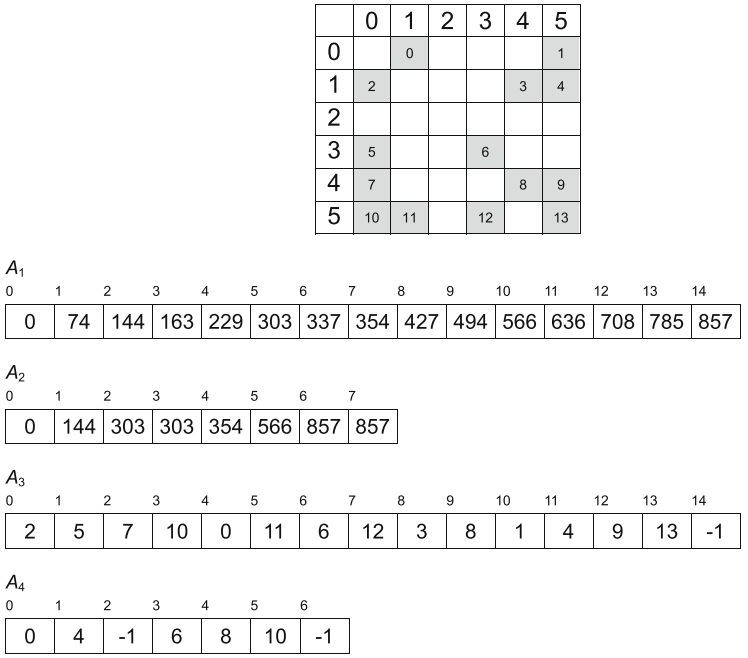

Fig. 3. Decomposition of the graph binary matrix. Shaded squares (top figure) denote non-empty boxes.

contain at least one set bit, hence the maximum gap between any two 1s in the resulting sequence is $2h - 1$ and thus for $h \leq 128$ each value can be stored on a byte (a preliminary experiment with $h = 256$ and using a byte code for gap encoding was rather unsuccessful). Fig. 2 presents the LM-bitmap variant.

Those two sequences: the compacted long list and the flag sequence (either raw, or gap-encoded) are then concatenated and compressed with the Deflate algorithm.

One can see that the key parameter here is the block size h. Using a larger h lets us exploit a wider range of similar lists but also has two drawbacks. The flag sequence gets more and more sparse (for example, for $h = 64$ and the EU-2005 crawl, as much as about 68% of its list indicators have only one set bit out of 64!) and the Deflate compressor is becoming relatively inefficient on those data; a drawback more important in the LM-bitmap variant. Worse, decoding larger blocks takes more time.

4. Our algorithm for 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. More sophisticated solutions are however known (cf. Section 2). They use 2D structures that support bidirectional navigation over the graph.

We propose a simple technique for this problem scenario. The algorithm, denoted as 2D in the experimental section, partitions the binary matrix of the 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 lets us attain the smallest space ever reported in the literature among the algorithms supporting bidirectional navigation, for example 1.741 bpe for EU-2005, at the average extraction time per link of 14.1 μs for forward neighbor queries and 13.9 μs for reverse neighbor queries (or respectively 315 and 310 μs per adjacency list). The access time to a neighbor list may be relatively long, but is still at least an order of magnitude better than a typical hard disk access time.

We present the algorithm on the example of the EU-2005 dataset. First, we partition the graph binary matrix M (where $n = 862664$ nodes) into boxes (squares) of size B (the boundary areas may be rectangular). Assume that $B = 1024$ (in the experiments we vary this parameter). The compressed content of non-empty (i.e. those that contain at least one edge) boxes, in row-major order, is stored in an archive C. We keep an array A_1 of 4-byte offsets for successive compressed non-empty boxes in C. Those boxes are compressed independently and details on that will be presented later. In our example there are about 24,700 non-empty boxes, hence the size of A_1 is just below 100 KB. The array A_2, of size $\lceil n/B \rceil$ stores the offsets from A_1 for the leftmost non-empty boxes in successive horizontal stripes. To enable reverse neighbor queries we also maintain arrays A_3 and A_4. The array A_3 enables to scan vertical stripes, by storing all A_1 indexes in column-major order (followed by value -1 as a terminator). Finally, A_4 stores the indexes from A_3 of the topmost non-empty boxes in successive vertical stripes. (Empty stripes, very rare of course, are marked with value -1.) Fig. 3 illustrates.

Now we present how forward and reverse neighbors of a given node are found.

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

Dataset	EU-2005	Indochina-2004	UK-2002	Arabic-2005
Nodes	862,664	7,414,866	18,520,486	22,744,080
Edges	19,235,140	194,109,311	298,113,762	639,999,458
Edges/nodes	22.30	26.18	16.10	28.14
% of empty lists	8.309	17.655	14.908	14.514
Longest list length	6985	6985	2450	9905

To find the forward neighbors of node j we must retrieve and decode all the non-empty boxes overlapping the jth row of the matrix, i.e. from the $\lfloor n/j \rfloor$-th vertical stripe. Note that for efficient retrieval we need only to find quickly in the array A_1 the pointer to first (leftmost) such box as all its successors will be pointed from the following cells of A_1. Now, let $x = A_2[\lfloor n/j \rfloor]$ and $A_1[x]$ is the pointer to the first desired box.

The difference $A_1[x+1] - A_1[x]$ tells the size of the compressed box, hence we can safely decompress it from C. The successive boxes in the vertical stripe are pointed by $A_1[x+1], A_1[x+2], \ldots$, and we know when to stop by examining $A_2[\lfloor n/j \rfloor + 1]$.

Decoding a box involves Deflate decompression (for some boxes however Deflate is not used, this usually concerns almost empty boxes) and optional box content transposition. The decoded boxes must be filtered to return only those values which belong to desired list j.

Similarly, reverse neighbors are obtained with aid of A_3 and A_4, but now the compressed boxes are non-contiguous in C.

The non-local access pattern may be harmful, but on the other hand, it appears that more boxes are Deflate-compressed after transposition than without it, which means that inverse transposition is not performed for obtaining reverse neighbors for those boxes. The net effect is that for the same dataset returning forward and reverse neighbors takes almost the same time on average.

Now we present the compression scheme for a box. 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; the 3-byte codeword is enough to handle $B \leq 4096$). 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 node requires decoding many boxes, even those that have no item in common with the desired neighbor list. For the EU-2005 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 totaling the same size, because of the locality of memory accesses. Also, as said, those average-case estimations are far from the worst case.

5. Experimental results

The experiments with the SSL algorithm comprise only the datasets EU-2005 and Indochina-2004, while the more practical LM and 2D variants are tested also on the UK-2002 and Arabic-2005 crawls; all the datasets are downloaded from the WebGraph project ([5], http://webgraph.dsi.unimi.it/). Note that we test the natural order versions of them, as using reordered variants (also available from the WebGraph project) may be more efficient but then the compression of the corresponding URL data deteriorates.

The main characteristics of those datasets are presented in Table 1.

The main experiments (Section 5.1) were run on a machine equipped with an Intel Core 2 Quad Q9450 CPU, 8 GB of RAM and Microsoft Windows XP (64-bit). All presented algorithms were implemented in Java and launched on the 64-bit JVM 7. For the benchmarks we have used source codes provided by our competitors, compiled and run in the same environment. 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, 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 first algorithm, SSL, has three parameters: the number of flags used (either 2 or 4, where 2 flags mimic the Boldi–Vigna scheme and 4 correspond to Fig. 1), the byte encoding scheme (either using 2 or 3 codeword lengths) and the residual block size threshold BSIZE, set to 8192 bytes (results with other BSIZE values can be found in [17]). The remaining parameters

Table 2
SSL algorithm, compression ratios in bits per edge, access times in μs.

	EU-2005				Indochina-2004			
	Direct		Transpose		Direct		Transpose	
	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time
2a	2.276	9.72	2.336	9.70	1.101	11.62	1.083	10.91
2b	2.190	10.08	2.282	9.72	1.062	12.25	1.061	12.11
4a	1.730	15.35	1.805	14.12	0.936	15.17	0.901	15.92
4b	1.692	15.72	1.778	14.63	0.907	15.89	0.887	17.12

Table 3
EU-2005 and Indochina-2004 datasets, direct graphs. Compression ratios in bits per edge and access times per edge, access times in μs. "LMx" stands for LM-bitmap with $h = x$. To the results of BV (7, 3) the amount of 0.510 bpe and 0.348 bpe should be added, respectively, corresponding to extra data required to access the graph in random order.

	EU-2005		Indochina-2004	
	bpe	Time	bpe	Time
BV (7, 3)	5.169	0.24	2.063	0.21
2a	2.276	9.72	1.101	11.62
4b	1.692	15.72	0.907	15.89
LM16	2.963	0.31	1.668	0.43
LM32	2.373	0.55	1.320	0.55
LM64	2.008	1.05	1.097	0.79

constitute four variants: (2a) two flags and two codeword lengths, (2b) two flags and three codeword lengths, (4a) four flags and two codeword lengths, (4b) four flags and three codeword lengths.

As expected, the compression ratios improve with using more flags and more dense byte codes (Table 2). Table 3 presents 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 216 to 304 μs for 2a and from 326 to 448 μs for 4b. This is an order of magnitude less than the access time of 10 K or 15 K RPM hard disks.

Our second algorithm, LM, has one parameter h which is the number of lines (lists) per block. We conducted experiments for $h = 16, 32, 64$. The results are presented in the last three rows of Table 3. For this comparison, only the LM-bitmap variant is used. We see that even LM64 cannot reach the compression of SSL 4b, but its list extraction is faster 15–20 times. The fastest of the variants presented here, LM16, is 1.3 and 2.0 times slower than BV (7, 3), respectively, with much better compression (we checked also LM8, only on EU-2005: the results are 3.814 bpe and 0.20 μs per edge).

A larger experiment was conducted on four datasets (in both direct and transposed versions). The obtained results are presented in Fig. 4. The LM-bitmap variant fares better in comparison with smaller blocks (h up to 16), but then the LM-diff variant starts to win in compression and the gap grows with growing h. Unfortunately, decoding LM-diff blocks is also in most cases costlier, with 74% maximum loss for Indochina-2004 direct, $h = 64$. On average, its loss in speed to LM-bitmap is not, however, that big.

Finally, we test the 2D algorithm with varying block sizes (Fig. 5). Compared to the most successful compression schemes with bidirectional query support [11,20], our algorithm is capable of reaching better compression ratios. It also achieves better space–time tradeoffs than the results [20] but the contrary is true in comparison with the scheme from [11], k^2-partitioned. This means that 2D is the leader for high compression, but when shorter access times are needed there exist better tradeoffs.

6. Conclusions

We presented three algorithms for Web graph compression, which achieve better compression results than those presented in the literature, although two of them for the price of relatively slow access time. Perhaps the most interesting algorithm, LM, is competitive to the best algorithms known from the literature. Our approach lets us achieve compression ratios not reported in the literature (LM-diff, 64 and 128), for one-directional queries, for moderate slow-down in list accesses (the best tradeoff here, however, seem to be the variants LM-diff and LM-bitmap for $h = 32$). If even better compression ratios are welcome then our SSL 4b variant can be considered, being more than an order of magnitude slower. We point out that one extreme tradeoff in succinct in-memory data structures is when accessing the structure is only slightly faster than reading data from disk (i.e. about 5–10 ms disk access time is the limit). We also presented a surprisingly simple algorithm with bidirectional neighbor support, 2D, which dominates in compression ratio over the competitors known from the literature. It is however slower than some recent proposals when more moderate compression is enough.

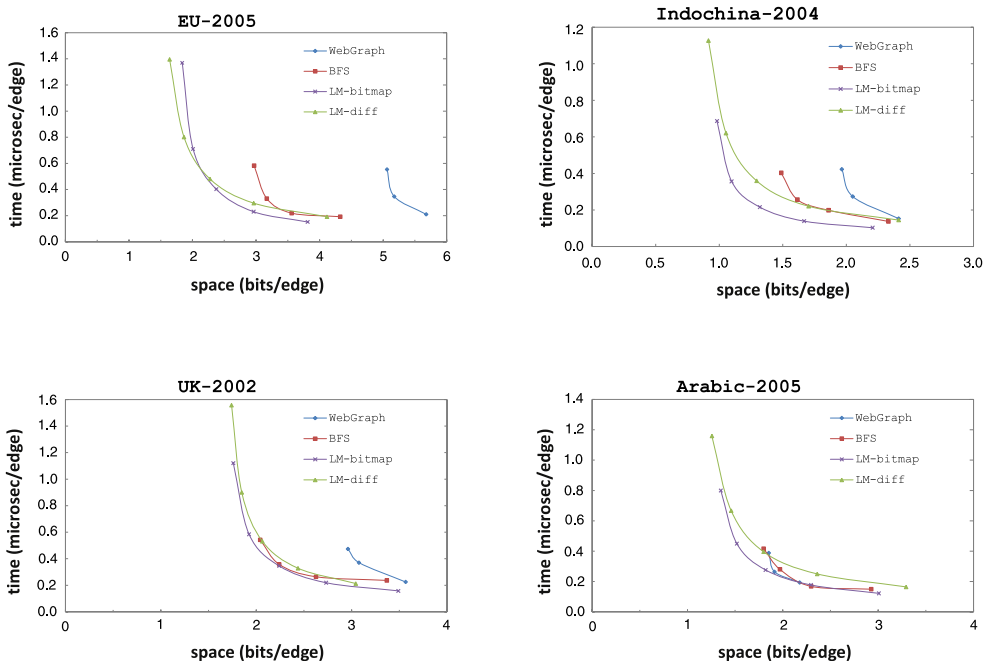

Fig. 4. Compression algorithms with one-directional queries. Compression ratios (bpe) and access times per edge. The values h for LM-bitmap and LM-diff are from {8, 16, 32, 64, 128}.

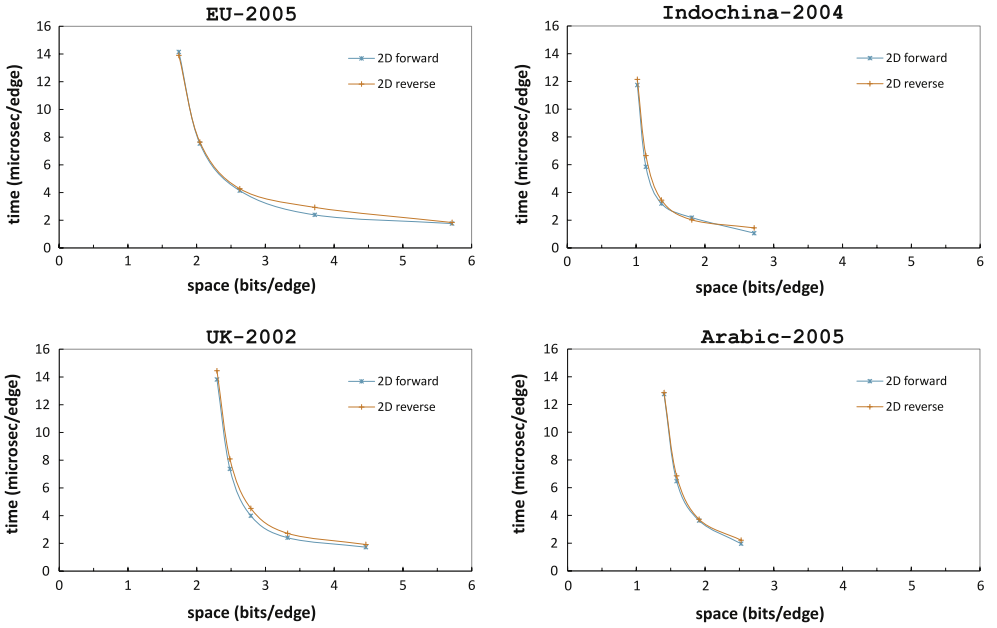

Fig. 5. 2D algorithm, compression ratios in bits per edge, access times for direct and reverse neighbors in μs. Block sizes from {64, 128, 256, 512, 1024}.

Our 1D algorithms work locally. In the future we are going to try to squeeze out some global redundancy while compressing the LM byproducts. Also exploring ways to make the 2D algorithm faster is within our interests.

Acknowledgment

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

References

[1] V.N. Anh, A.F. Moffat, Local modeling for webgraph compression, in: J.A. Storer, M.W. Marcellin (Eds.), DCC, IEEE Computer Society, 2010, p. 519.
[2] A. Apostolico, G. Drovandi, Graph compression by BFS, Algorithms 2 (3) (2009) 1031–1044.
[3] Y. Asano, Y. Miyawaki, T. Nishizeki, Efficient compression of web graphs, in: X. Hu, J. Wang (Eds.), COCOON, in: Lecture Notes in Computer Science, vol. 5092, Springer, 2008, pp. 1–11.
[4] K. Bharat, A.Z. Broder, M.R. Henzinger, P. Kumar, S. Venkatasubramanian, The connectivity server: fast access to linkage information on the web, Computer Networks 30 (1–7) (1998) 469–477.
[5] P. Boldi, B. Codenotti, M. Santini, S. Vigna, UbiCrawler: a scalable fully distributed web crawler, Software: Practice and Experience 34 (8) (2004) 711–726.
[6] P. Boldi, M. Rosa, M. Santini, S. Vigna, Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks, in: S. Srinivasan, K. Ramamritham, A. Kumar, M.P. Ravindra, E. Bertino, R. Kumar (Eds.), WWW, ACM, 2011, pp. 587–596.
[7] P. Boldi, M. Santini, S. Vigna, Permuting web and social graphs, Internet Mathematics (2009) 257–283.
[8] P. Boldi, S. Vigna, The webgraph framework I: compression techniques, in: S.I. Feldman, M. Uretsky, M. Najork, C.E. Wills (Eds.), WWW, ACM, 2004, pp. 595–602.
[9] N. Brisaboa, S. Ladra, G. Navarro, K2-trees for compact web graph representation, in: Proc. 16th International Symposium on String Processing and Information Retrieval, SPIRE, in: LNCS, vol. 5721, Springer, 2009, pp. 18–30.
[10] G. Buehrer, K. Chellapilla, A scalable pattern mining approach to web graph compression with communities, in: M. Najork, A.Z. Broder, S. Chakrabarti (Eds.), WSDM, ACM, 2008, pp. 95–106.
[11] F. Claude, S. Ladra, Practical representations for web and social graphs, in: C. Macdonald, I. Ounis, I. Ruthven (Eds.), Proc. ACM Conference on Information and Knowledge Management, ACM, 2011, pp. 1185–1190.
[12] F. Claude, G. Navarro, Fast and compact web graph representations, Tech. Rep. TR/DCC-2008-3, Department of Computer Science, University of Chile, April 2008.
[13] F. Claude, G. Navarro, Fast and compact web graph representations, ACM Transactions on the Web (TWEB) 4 (4) (2010) article 16.
[14] F. Claude, G. Navarro, Extended compact web graph representations, in: T. Elomaa, H. Mannila, P. Orponen (Eds.), Algorithms and Applications, in: Lecture Notes in Computer Science, vol. 6060, Springer, 2010, pp. 77–91.
[15] J.S. Culpepper, A. Moffat, Enhanced byte codes with restricted prefix properties, in: M.P. Consens, G. Navarro (Eds.), SPIRE, in: Lecture Notes in Computer Science, vol. 3772, Springer, 2005, pp. 1–12.
[16] R.F. Geary, N. Rahman, R. Raman, V. Raman, A simple optimal representation for balanced parentheses, in: S.C. Sahinalp, S. Muthukrishnan, U. Dogrusöz (Eds.), Proceedings Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, in: Lecture Notes in Computer Science, vol. 3109, Springer-Verlag, 2004, pp. 159–172.
[17] S. Grabowski, W. Bieniecki, Tight and simple web graph compression, in: J. Holub, J. Žďárek (Eds.), Proc. Prague Stringology Conference, 2010, pp. 127–137.
[18] S. Grabowski, W. Bieniecki, Merging adjacency lists for efficient Web graph compression, in: Man-Machine Interactions 2, in: Advances in Intelligent and Soft Computing, vol. 103, Springer, 2011, pp. 385–392.
[19] X. He, M.-Y. Kao, H.-I. Lu, A fast general methodology for information-theoretically optimal encodings of graphs, SIAM Journal on Computing 30 (3) (2000) 838–846.
[20] C. Hernández, G. Navarro, Compression of web and social graphs supporting neighbor and community queries, in: Chid Apté, Joydeep Ghosh, Padhraic Smyth (Eds.), Proc. 5th ACM Workshop on Social Network Mining and Analysis, SNA-KDD, ACM, 2011.
[21] G. Jacobson, Succinct static data structures, Ph.D. Thesis, 1989.
[22] S. Ladra, Algorithms and compressed data structures for information retrieval, Ph.D. Thesis, 2011.
[23] N.J. Larsson, A. Moffat, Off-line dictionary-based compression, Proceedings of the IEEE 88 (11) (2000) 1722–1732.
[24] J.I. Munro, 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.
[25] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Computing Surveys 39 (1) (2007) article 2.
[26] P. Procházka, J. Holub, New word-based adaptive dense compressors, in: J. Fiala, J. Kratochvíl, M. Miller (Eds.), IWOCA, in: Lecture Notes in Computer Science, vol. 5874, Springer, 2009, pp. 420–431.
[27] K. Randall, R. Stata, R. Wickremesinghe, J. Wiener, The link database: fast access to graphs of the web, 2001.
[28] G. Turán, On the succinct representation of graphs, Discrete Applied Mathematics 15 (2) (1984) 604–618.