Merging Adjacency Lists for Efficient Web Graph Compression

Szymon Grabowski and Wojciech Bieniecki

Abstract. Analysing Web graphs meets a difficulty in 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. Our algorithm belongs to this category. It works on contiguous blocks of adjacency lists, and its key mechanism is merging the block into a single ordered list. This method achieves compression ratios much better than most methods known from the literature at rather competitive access times.

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 [15], dictionaries, trees and graphs [14]. 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.

One particular huge object of significant interest is 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

Szymon Grabowski · Wojciech Bieniecki
Computer Engineering Department, Technical University of Łódź,
al. Politechniki 11, 90-924 Łódź, Poland
e-mail: `{sgrabow,wbieniec}@kis.p.lodz.pl`

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.

As of May 2011, it is estimated that Google's index has about 32 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 3.2 TB of memory, ways beyond the capacities of the current RAM memories.

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 (e.g. planar ones). The first papers dedicated to Web graph compression, which appeared around 2000, pointed out some redundancies in the graph, e.g., that successive adjacency lists tend to have nodes in common, if they are sorted in URL lexicographical order, but they failed to achieve impressive compression ratios.

One of the most efficient (and most often used as a reference in newer works) compression schemes for Web graph was presented by Boldi and Vigna [6] in 2003. Their method (BV) is likely to achieve around 3 bpe (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. Based on WebGraph datasets [4], 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 redundancy, one bit per entry on the referenced list is used, to denote which of its integers are copied to the current list, and which are not. Those bit-vectors tend to contain runs of 0s and 1s, and thus are compressed with a sort of RLE (run-length encoding). The integers on the current list which didn't occur on the referenced list are stored too; intervals of consecutive integers are also encoded in an RLE manner, while the numbers which do not fall into any interval (*residuals*) are differentially encoded. Finally, the BV 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.

Claude and Navarro [11] took a totally different approach of grammar-based compression. In particular, they focus on rule-based Re-Pair [13] and dictionary-based 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.

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

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 unbounded reference chains (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 several types of blocks in the matrix. 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, from 2.3 to 28.7 milliseconds (Core2 Duo E6600 2.4 GHz, Java implementation), depending on a dataset: the longest time is even longer than hard disk access time! 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 [9], 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 [10] they should be rather competitive and at least much shorter than of the algorithm from [3], with 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 twice faster than returning the whole neighbor list. Still, we note that using non-lexicographical ordering is probably harmful for compact storing of the webpage URLs themselves (a problem accompanying pure graph structure compression in most practical applications).

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.

Some recent works focus on graph compression with support for bidirectional navigation [8, 10] and experiments show that this approach 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 microseconds (Pentium4 3.0 GHz).

The smallest compressed Web graph datasets (only EU-2005 and Indochina-2004 used in the experiments) were reported by Grabowski and Bieniecki [12]; their best results were about 1.7 bpe and 0.9 bpe (including offsets to compressed chunk beginnings) for those graphs, respectively, which is 2.5–3 times less than from BV variant with fast access. The algorithm (called SSL for 'Similarity of Successive Lists') exploits similar ideas to BV, but uses Deflate (zip) compression on chunks of byproducts at its last phase. Unfortunately, the price for those record-breaking compression ratios is random list access time, two orders of magnitude longer than in BV.

3 Our Algorithm

We present an algorithm (Alg. 1, LM stands for 'List Merging') that works locally, in blocks having the same number of adjacency 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: differentially encoded, zero-terminated and submitted to a byte coder, using 1, 2 or b bytes per codeword, where b is the smallest number of bytes sufficient to handle any node number in a given graph (in practice, this means $b = 4$ except for the smallest dataset, EU-2005, where $b = 3$ was enough).

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.

Those two sequences, the compacted long list and the (raw) flag sequence, are 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 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. Worse, decoding (including decompression) larger blocks takes longer time.

Alg. 1 GraphCompressLM(G,h).

```
$outF \leftarrow [\,]$
$i \leftarrow 1$
for $line_i, line_{i+1}, \ldots, line_{i+h-1} \in G$ do
    $tempLine_1 \leftarrow line_i \cup line_{i+1} \cup \ldots \cup line_{i+h-1}$
    $tempLine_2 \leftarrow$ removeDuplicates($tempLine_1$)
    $longLine \leftarrow$ sort($tempLine_2$)
    $items \leftarrow$ diffEncode($longLine$) + [0]
    $outB \leftarrow$ byteEncode($items$)
    for $j \leftarrow 1$ to $|longLine|$ do
        $f[1 \ldots |longLine|] \leftarrow [0,0,\ldots,0]$
        for $k \leftarrow 1$ to $h$ do
            if $longLine[j] \in line_{i+k-1}$ then $f[k] \leftarrow 1$
        append($outF$, bitPack($f$))
    compress(concat($outB$, $outF$))
    $outF \leftarrow [\,]$
    $i \leftarrow i+h$
```

4 Experimental Results

We conducted experiments on the crawls EU-2005, Indochina-2004 and UK-2002 [4], downloaded from the WebGraph project.[2] The main characteristics of those datasets are presented in Table 1.

Table 1 Selected characteristics of the datasets used in the experiments

Dataset	EU-2005	Indochina-2004	UK-2002
Nodes	862,664	7,414,866	18,520,486
Edges	19,235,140	194,109,311	298,113,762
Edges / nodes	22.30	26.18	16.10
% of empty lists	8.31	17.66	14.91
Longest list length	6985	6985	2450

The main experiments 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.

[2] `http://webgraph.dsi.unimi.it/`

We tested the following algorithms:

- The Boldi and Vigna algorithm [6], variant $(7,3)$, i.e., the sliding window size is 7 and the maximum reference count 3,
- The Apostolico and Drovandi algorithm [2], using BFS webpage ordering, with parameter l (the number of nodes per compressed block) set to $\{4,8,16,32,1024\}$ and parameter r (the root of the BFS) set to 0,
- The variant offering strongest compression from our earlier work [12], SSL 4b,
- Our algorithm (LM) from this work, with 8, 16, 32 and 64 lists per chunk.

We used the implementations publicly available from the authors of the respective algorithms. Note that all those implementations were written in Java, which makes the comparison fair.

Table 2 Comparison of Web graph compressors. Compression ratios in bits per edge and average access times per edge are presented. Offset data are included

Dataset	EU-2005		Indochina-2004		UK-2002	
	bpe	time [μs]	bpe	time [μs]	bpe	time [μs]
BV (7,3)	5.679	0.227	2.411	0.181	3.567	0.262
BFS, l4	4.325	0.242	2.331	0.147	3.369	0.307
BFS, l8	3.561	0.227	1.860	0.179	2.627	0.260
BFS, l16	3.169	0.351	1.615	0.264	2.242	0.343
BFS, l32	2.969	0.617	1.488	0.420	2.042	0.542
BFS, l1024	2.776	15.425	1.363	9.979	1.851	12.338
SSL 4b	1.692	22.276	0.907	23.521	1.678	23.654
LM8	3.814	0.179	2.207	0.136	3.490	0.196
LM16	2.963	0.265	1.668	0.166	2.733	0.253
LM32	2.373	0.453	1.320	0.252	2.241	0.395
LM64	2.008	0.815	1.097	0.429	1.925	0.654

Several conclusions can be drawn from the results. BFS and LM seem to be the best choices, considering the tradeoff between space and access time. When access time is at a premium, those two are comparable, with a slight advantage of LM (with 8 or 16 lists, confronted with BFS $-l4$ or $-l8$). When stronger compression is required, LM reaches bpe results rather inaccessible to BFS, with the exception of the UK-2002 dataset. In the latter case, the BFS –l1024 archive is by 4 % smaller than LM64 archive, for the price of 19 times longer average access time.

By default, BFS is a randomized algorithm and there are minor yet noticable differences in its produced compressed graph sizes. Fixing the parameter r makes the results deterministic. To avoid guessing, we simply set it to 0.

The oldest algorithm, BV, may seem slightly dated, but we note the work [5] from the same team, where they showed that non-URL based ordering can lead to compressed Web graphs about 10 % smaller using their old baseline scheme (in a practical variant), and even up to 35 % smaller in case of transposed graphs. BFS plays in the same league and reordering of nodes is its core feature. It should be stressed that using a different ordering than by URLs arranged lexicographically may spoil the compression of URLs themselves; a practically important but oft-neglected factor (a recent work pointing out this issue with some solutions tested is [7]).

As expected, our earlier algorithm, SSL 4b, remains the strongest but also definitely the slowest competitor. It also uses Deflate compression in its final phase. We note that the results of SSL 4b, nevertheless, are somewhat better (mostly in access time but also slightly in compression) than in our previous paper, which is due to removing some inefficiency in its Deflate invocation. BV (7,3) timings are also better than in our previous tests on the same machine and using the same methodology, a fact for which we do not find a good explanation. Perhaps this might be due to an update of JDK 6 version.

Finally, we replaced Deflate in our LM algorithm with LZMA[3], known as one of the strongest LZ77-style algorithms. Unfortunately, we were disappointed: only with chunk size of 64 lists LZMA proved better than Deflate (from 4 % to 6 %) but the access times were more than 3 times longer. With smaller chunks, the Deflate algorithm was usually better in compression (the smaller chunks the greater its advantage) while the access times revealed the same pattern as above.

5 Conclusions

We presented a surprisingly simple yet effective Web graph compression algorithm, LM. Varying a single and very natural parameter (chunk size, in the number of lists) we can obtain several competitive space-time tradeoffs. As opposed to some other alternatives (in particular, BFS), LM does not reorder the graph. Still, it could be quite interesting to run LM over a permuted graph, making use of the conclusions drawn in [5].

Our algorithm works locally. In the future we are going to try to squeeze out some global redundancy while compressing the LM byproducts. A natural candidate for such experiments is the RePair algorithm [13, 11]. Other lines of research we are planning to follow are Web graph compression with bidirectional navigation and efficient compression of URLs.

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

References

1. Anh, V.N., Moffat, A.F.: Local modeling for webgraph compression. In: Storer, J.A., Marcellin, M.W. (eds.) Proceedings of the Data Compression Conference (DCC), p. 519. IEEE Computer Society, Los Alamitos (2010)

[3] `http://sourceforge.net/projects/sevenzip/files/LZMA%20SDK/lzma920.tar.bz2`

2. Apostolico, A., Drovandi, G.: Graph compression by BFS. Algorithms 2(3), 1031–1044 (2009)
3. Asano, Y., Miyawaki, Y., Nishizeki, T.: Efficient compression of web graphs. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 1–11. Springer, Heidelberg (2008)
4. Boldi, P., Codenotti, B., Santini, M., Vigna, S.: UbiCrawler: A scalable fully distributed web crawler. Software: Practice & Experience 34(8), 711–726 (2004)
5. Boldi, P., Santini, M., Vigna, S.: Permuting web and social graphs. Internet Mathematics 6(3), 257–283 (2010)
6. Boldi, P., Vigna, S.: The webgraph framework I: Compression techniques. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) Proceedings of the 13th International World Wide Web Conference, pp. 595–602. ACM Press, New York (2004)
7. Brisaboa, N., Cánovas, R., Claude, F., Martínez-Prieto, M., Navarro, G.: Compressed string dictionaries. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 136–147. Springer, Heidelberg (2011)
8. Brisaboa, N.R., Ladra, S., Navarro, G.: K2-trees for compact web graph representation. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 18–30. Springer, Heidelberg (2009)
9. Buehrer, G., Chellapilla, K.: A scalable pattern mining approach to web graph compression with communities. In: Najork, M., Broder, A.Z., Chakrabarti, S. (eds.) Proceedings of the International Conference on Web Search and Web Data Mining (WSDM), pp. 95–106. ACM, New York (2008)
10. Claude, F., Navarro, G.: Extended compact web graph representations. In: Elomaa, T., Mannila, H., Orponen, P. (eds.) Ukkonen Festschrift 2010. LNCS, vol. 6060, pp. 77–91. Springer, Heidelberg (2010)
11. Claude, F., Navarro, G.: Fast and compact web graph representations. ACM Transactions on the Web (TWEB) 4(4), 16:1–16:16 (2010)
12. Grabowski, S., Bieniecki, W.: Tight and simple Web graph compression. In: Holub, J., Žd'árek, J. (eds.) Proceeding of the Prague Stringology Conference, pp. 127–137 (2010)
13. Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. Proceedings of the IEEE 88(11), 1722–1732 (2000)
14. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses, static trees and planar graphs. In: Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), pp. 118–126. IEEE Computer Society, Los Alamitos (1997)
15. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1) (2007)