AUTOMATYKA • 2011 • Tom 15 • Zeszyt 3

Szymon Grabowski*, Jakub Swacha**

Compact Representation of URL Collections with Fast Access

1. Introduction

Efficient representation of a string dictionary is a well-known problem with applications e.g. in web searchers and spellchecking. Traditionally, the dictionary is relatively minor compared to the text from which the terms (words) are collected. Those applications usually arise in natural language (NL) processing, where the words apply to the famous Heaps' law: the dictionary size is $\Theta(n\beta)$, where n is the text size (in words) and β is usually between 0.4 and 0.6. In other words, the dictionary size grows roughly proportionally to the square root of the text length. This empirical law holds, when the growing text is in a single language (or, if not, the number of different languages in the text is considered constant). Moreover, experimental validations of this law were usually conveyed on rather clean, homogeneous text collections, e.g., articles from a newspaper archive.

When we shift our interest from "closed" homogeneous collections of rather high quality to the web crawls, we will notice, that the dictionaries built by indexers in various information retrieval tasks (the most canonical of which seem to be the extremely popular web searchers) can grow rather fast and the number of distinct terms can be on the order of 10^8 or more (http://lemurproject.org/clueweb09/). The reasons behind this include using various languages, numerous names (people, brands, product numbers, geographical names etc.), neologisms and e-speak (according to [1], about 20% of all *query* words in web searchers are non-dictionary words, and misspellings, for which matches in the documents are generally unlikely, are only a small fraction of them), and sheer typos.

We have just argued that even in NL processing the vocabularies can be large enough not to fit (together with auxiliary data structures) in the RAM memory of commodity PCs. Still, there are other applications, where the term collections can be huge. They include, e.g., bioinformatics and web graphs. We focus on the latter case.

* Technical University of Lodz, Computer Engineering Dept., Lodz, Poland
** University of Szczecin, Institute of Information Technology in Management, Szczecin, Poland

The web graph is a collection of nodes (web documents) and arcs (hyperlinks between them), obtained as a result of crawling. Typically those are only relatively small parts of the whole web, which is estimated to have tens of billions nodes (http://www.worldwidewebsize.com) and is highly dynamic. Web mining tasks are often based on studying the web graph topology, for ranking pages (e.g., with PageRank), spam detection, finding communities etc. To fit possibly large web crawls in the main memory, various web graph compression algorithms have been designed [2–5], providing fast access to adjacency lists. Still, the works on web graph compression usually ignore the representation of the URLs themselves, which in practice are taken into account for more robust mining.

The two standard queries to ordered set[1] (not necessarily with string keys) of size n are: *extract*(i), which returns i-th key in the collection, and *locate*(k), which return the index ($0\ldots n–1$) of key k in the set, if it exists, and –1 otherwise.

In this work we present an efficient compression algorithm for collections of URLs, supporting extract queries. In the design process we assumed (although this is not a requirement for our scheme) that the input URLs are lexicographically sorted, hence the standard front coding technique may be applied. The scheme support the extract query.

2. Related work

The literature on string dictionary data structures is ample but surprisingly few experimental works have dealt with non-NL (non-word) vocabularies. Although tries, binary or ternary search trees, splay trees or plain hash arrays are all classic solutions (a review can be found in [6]), capable to handle string collections of various characteristics, they are not succinct and large collections may not be possible to maintain in the main memory. A more compact solution is the minimum finite state automaton from [7] (in [8] it was shown how to attach satellite data to keys, e.g. the URL index (id) necessary in web graph analyses). Apart from several NL vocabularies (mostly Ispell lexicons for various languages), this algorithm has also been tested (among others) [7] on the collection of pathnames from ftp://ftp.funet.?/pub/.

In [9] (where another string dictionary data structure is introduced, the burst trie), experiments are run on relatively large web document collections (where the terms are, roughly, NL words), but also on q-grams from genomic and music data. In [6] and [10] the test collection comprises, among others, a dataset of almost 10M URLs, totaling 319 MB in plain textual form. The URLs are taken over many websites, but seemingly only small subsets of those websites have been recorded (it is hard to verify this guess since most of those

[1] Now we use the term (ordered) "set" (as opposed to "dictionary") to stress that the elements of the collections have no satellite data attached, and their keys' corresponding "values" are simply their indexes in the given order. In information retrieval / natural language processing the word "dictionary" is often used in the meaning of a "set" (of strings) in algorithmics.

sites no longer exist today). Moreover, many URLs in the collection occur several times (the number of distinct URLs is ~1.3M) hence the data cannot correspond to any real web crawl.

Interesting theoretical results concerning minimal monotone perfect hashing have been given in [11] and algorithmically engineered and tested in [12]. In those works, succinct order-preserving hash functions were given, assuming an existing (not imposed) lexicographical order of keys in an input collection. One of the experimental results [12] is that it is enough to spend about 6.5 bits per key on average for a 106M-key URL dataset (uk-2007-05), with fast access (about 30 μs per key) *apart from the keys themselves*[2]. This scheme has also been tested with Hu–Tucker (HT) encoding [13] of the input collection, reducing thus the overall space use. HT coding is in the same class of codes as Huffman coding, and is optimal among those codes that preserve lexicographical order of keys (which enables binary search directly over compressed data), while Huffman is optimal without this extra requirement.

A recent empirical study [14] of compact representations of string dictionaries (datasets of words, DNA *q*-grams, URLs and URIs) reveals that, especially in case of URLs, two techniques are particularly efficient, if both space and access time (locate and extract queries supported) are taken into account, namely grammar-based Re-Pair [15] and the folklore technique *front coding* (FC) accompanied with Hu–Tucker coding of the remaining suffixes.

3. Proposed algorithm

The proposed algorithm is extremely simple and somewhat similar to the FC + HT solution from [14]. The front coding technique is to replace the prefix of the current element (URL) shared with the previous element with its 1-byte length (in this way, common prefixes are limited to 255 characters, which is usually a negligible limitation), and the prefix lengths may be sent to a separate stream. The common prefix is not removed from every b_p-th line, to allow searching. This also means that the number of prefix lengths stored for a block of size b_p is $b_p - 1$.

We also considered removing shared URL suffixes (e.g., ".html"), but applying this idea (analogous to front coding) was beneficial for compression ratio only on a single dataset (eu-2005), hence we gave it up.

For the remaining stream of data (without the prefixes) we resign from Hu–Tucker coding, which works on characters, but first replace the common phrases in strings with 1-byte characters from 128…254 range (not used on URL characters) and then apply Deflate (zip) compression in blocks of *b* lines. The common phrases in many cases are frequent

[2] If the keys (URLs) themselves are not needed, the average 6.5 bits per key is enough to map each key to its lexicographical position. Still, some web analyses require the knowledge of the URLs.

only locally, hence we find them in superblocks of *sb* lines, requiring that *sb* is a multiple of *b*. One set of phrases per superblock is written (uncompressed, since it will be kept in memory during the search, with characters #255 as separators). For convenience we also require that b_p is a multiple of *b*. An exemplary set of parameters b_p, *b*, *sb* could be: 256, 64, 2048.

How to choose a "good" set of phrases is a hard combinatorial problem; we make it a bit easier considering only URL segments between the separators described with the following regular expression (class of characters): [.&=/_-]. Moreover, we set the minimum phrase length to 2.

Example: in the line http://www.skwigly.co.uk/banner/abmc.asp?b=62&z=45 there are the following potential phrases (separated with bars): http: | www | skwigly | co | uk | banner | abmc | 62 | 45 ("b" and "z" are eliminated as being too short). Please note that front coding is also likely to remove "http://" or "http://www." first.

We choose the 127 most frequent phrases in the superblock and replace them with 1-byte symbols, as described.

4. Experimental results

The experiments were run on three URL datasets available from the WebGraph project (http://webgraph.dsi.unimi.it/) [16]: eu-2005, indochina-2004 and uk-2002, sorted lexicographically[3]. See Table 1 for details.

Table 1
Datasets used in the experiments

Dataset	URLs [M]	Size [MB]	Avg prefix length [chars]	Avg suffix length [chars]
eu-2005	0.863	72.540	57.55	9.43
indochina-2004	7.414	642.708	55.65	9.41
uk-2002	18.520	1438.713	50.03	7.69

The implementation was written in C#, using MS Visual Studio 2008 environment, and the test machine was equipped with an AMD Phenom II 1075T 3.0 GHz (3.5 GHz with

[3] The WebGraph site stores the graphs in two variants: in natural order (corresponding to URLs sorted lexicographically) and in permuted order. Our tests use the natural-order versions (with -nat in the filenames). The uk-2002 dataset downloaded for the experiments in [14] was the permuted version but their dictionaries first sort (in lexicographic order) all URLs in the dataset, so our experiments on uk-2002 are compatible. (Miguel A. Martínez-Prieto, priv. corr., June 2011).

Turbo Core), 8 GB of RAM, running 64-bit Windows 7. The extraction algorithm works in four phases: (*i*) extracts the phrase dictionary for the superblock to which the given block belongs, (*ii*) decodes the prefix lengths in a given block, (*iii*) decompresses the prefix-truncated URL data, and (*iv*) recovers the full URLs one by one in the given block up to the queried entry.

The amount of parameter combinations is obviously too large to be tested exhaustively. We chose $sb = 4096$ for all the tests, and $b_p = 256$ in almost all tests (the only exception was the case of $b = 512$, when b_p was set to the same value), varying the block size b. Table 2 presents the detailed results, including average (over 100,000 randomly selected index values in the range of the number of URLs in a given dataset) extract time per URL.

Table 2
Compression and extract time results

Dataset	b	b_p	Archive size [B]	Bytes / URL	Avg extract time [µs]
eu-2005	32	256	4269043	4.95	85.7
eu-2005	64	256	3366337	3.90	102.7
eu-2005	128	256	2834634	3.29	133.4
eu-2005	256	256	2505860	2.90	182.9
eu-2005	512	512	2239830	2.60	285.9
indochina-2004	32	256	43583128	5.88	96.5
indochina-2004	64	256	35816287	4.83	119.3
indochina-2004	128	256	31167905	4.20	155.0
indochina-2004	256	256	28293330	3.82	215.6
indochina-2004	512	512	25811296	3.48	336.9
uk-2002	32	256	125205094	6.76	108.2
uk-2002	64	256	105486502	5.70	134.1
uk-2002	128	256	93479010	5.05	174.8
uk-2002	256	256	85724403	4.63	241.0
uk-2002	512	512	79369434	4.29	376.0

As one can see, doubling the block size results in significantly less than doubling the extract time, which may be due to e.g. decompression model startup delay. We can compare our results to those from Brisaboa et al. [14] only on uk-2002. Their most compact solution, XBW + RRR, achieves about 134 MB ($M = 10^6$) at extract time over 600 µs on an Intel C2D 3.16 GHz. Still, the second most compact scheme, RePair, is much more attractive in terms of space-time tradeoff, achieving less than 5 μs (!) extract time and its archive size is about 179 MB. All the schemes tested in [14] handle both extract and locate queries, while our solution so far supports only extract queries (essentially, there should not be a problem to enhance it with this functionality, but at a cost of some compression loss). We also admit the solutions in [14] are general-purpose dictionaries while the phrase restriction in our algorithm is adapted to URLs.

We note that URL extraction times on the order of 200–300 μs are at least 20 times longer than extraction times for a node's adjacency list in the LM graph compression algorithm [5]. Still, we also believe that accessing URLs is relatively infrequent in web graph analyses (those data may be retrieved after finding some interesting subgraph first).

5. Conclusions

Compression of URL collections with fast access is a special case of the classic problem of efficient string dictionary representation. Although the general problem is very old and has rich literature, huge collections of URLs appeared in public repositories relatively recently. The motivation for handling such data comes from web graph analyses.

In this work we presented a very simple, yet compact scheme for representing URL collections, with support for the extract query. The archive consists of superblocks, for which dictionaries of (locally) frequent phrases are kept, and blocks. To decode *i*-th URL we need to extract its corresponding phrase dictionary and then its block. Much of the compression is achieved from using the Deflate compression algorithm, known from its high decompression speed. Still, we are not quite happy with the achieved extract times and future work should mostly focus on speed optimizations. We also plan to add support for locate queries.

Acknowledgements

The work was partially supported by the Polish Ministry of Science and Higher Education under the project N N516 477338 (2010–2011) (1st author).

References

[1] Ahmad F., Kondrak G., *Learning a spelling error model from search query logs*. Proc. Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Vancouver, British Columbia, Canada, 2005, 955–962.

[2] Boldi P., Vigna S., *The webgraph framework I: compression techniques*. WWW 2004, 595–602.

[3] Apostolico A., Drovandi G., *Graph compression by BFS*. Algorithms, 2(3), 2009, 1031–1044.

[4] Claude F., Navarro G., Fast *and compact web graph representations*. ACM Transactions on the Web (TWEB), 4(4):article 16, 2010.

[5] Grabowski Sz., Bieniecki W., *Merging adjacency lists for efficient Web graph compression*. Accepted to ICMMI, 2011.

[6] Askitis N., *Efficient Data Structures for Cache Architectures*. Ph.D. dissertation, RMIT University, Melbourne, Victoria, Australia, 2007.
[7] Ciura M.G., Deorowicz S., *How to squeeze a lexicon*. Software—Practice and Experience 31(11), 2001, 1077–1090.
[8] Skibiński P., Grabowski Sz., Deorowicz S., *Revisiting dictionary-based compression*. Software–Practice and Experience, 35(15), 2005, 1455–1476.
[9] Heinz S., Zobel J., Williams H. E., *Burst tries: a fast, efficient data structure for string keys*. ACM Transactions on Information Systems, 20(2), 2002, 192–223.
[10] Askitis N., Sinha R., *HAT-trie: A Cache-conscious Trie-based Data Structure for Strings*. ACSC 2007: 97–105.
[11] Belazzougui D., Boldi P., Pagh R., Vigna S., *Monotone minimal perfect hashing: searching a sorted table with O(1) accesses*. SODA, 2009, 785–794.
[12] Belazzougui D., Boldi P., Pagh R., Vigna S., *Theory and Practise of Monotone Minimal Perfect Hashing*. ALENEX, 2009, 132–144.
[13] Hu T., Tucker A., *Optimal computer search trees and variable-length alphabetical codes*. SIAM Journal on Applied Mathematics, 21(4), 1971, 514–532.
[14] Brisaboa N., Cánovas R., Claude F., Martínez-Prieto M.A., Navarro G., *Compressed String Dictionaries*. SEA, 2011, 136–147.
[15] Larsson N.J., Moffat J.A., *Offline dictionary-based compression*. Proc. of the IEEE, 88, 2000, 1722–1732.
[16] Boldi P., Codenotti B., Santini M., Vigna S., *UbiCrawler: A scalable fully distributed web crawler*. Software–Practice and Experience, 34(8), 2004, 711–726.