DCA using Suffix Arrays

Martin Fiala Jan Holub
Department of Computer Science and Engineering, Czech Technical University in Prague,
Karlovo náměstí 13, 121 35, Prague 2, Czech Republic, e-mail: holub@fel.cvut.cz

DCA (Data Compression using Antidictionaries) is a novel lossless data compression method working on bit streams presented by Crochemore et al. [1]. DCA takes advantage of words that do not occur as factors in the text, i.e. that are forbidden. Due to these forbidden words (antiwords), some symbols in the text can be predicted. First an input text (over binary alphabet $\Sigma = \{0, 1\}$) is analyzed and all minimal forbidden words are found and stored into antidictionary AD. Whenever we reach a word u so that $ua \in AD$, $u \in \Sigma^*, a \in \Sigma$, the following symbol can be predicted as a complement of a and does not need to be stored. When compressing the file, symbols that can be predicted are erased. Once we have the antidictionary constructed the compression as well as the decompression are extremely fast—a simple transducer is used.

Suffix Array Usage In [1] the suffix trie was used for the antidictionary construction, which is the most time and space consuming part of the method. One of the main problems of suffix trie is its memory consumption. Even for antiwords longer than 30 bits and small input files, the suffix trie size grows very fast and needs tens to hundreds megabytes. Creation and traversal through the whole trie is quite slow. We build the antidictionary using suffix array in time $\mathcal{O}(k * N \log N)$, where k is maximal antiword length. Length of suffix array and LCP constructed over the binary alphabet will be 8 times length of the input text. Still memory requirements for suffix array and LCP construction depend only on the length N of input text with $\mathcal{O}(N)$, instead of suffix trie with exponential complexity depending on the trie depth. We use Manzini-Ferragina construction [3] of suffix array.

Dynamic Compression Scheme With dynamic approach we read text only once, we compress input and modify AD at the same time. Whenever we read some input, we recompute AD again and use it for compressing the next input. Every time we read a symbol that violates the current AD and brings a forbidden word, we need to handle this exception. In the approach we don't have to separately encode AD, do self-compression or even to simple prune it. We simply use all antiwords found yet. Memory requirements are smaller, method is quite fast, as it does not need to do breadth first search for building AD or any other tree traversal for computing gains. Even it is very simple to implement it if we don't bother with suffix trie memory greediness and we don't need to read the text twice as in the static scheme. On the other hand there are some disadvantages as well: decompression is slower, parallel compressors/decompressors cannot be used, we lose k-local property. We have implemented static and dynamic methods with RLE (run length encoding) as well as almost antiwords.

file	original	gzip	bzip2	almostaw-30	dyn.-rle-32	rle-34	DI [2]
news	377109	144835	**118600**	173141	164269	180022	214986
obj1	21504	**10318**	10787	16154	13160	14544	18528
pic	513216	56438	49759	34159	**24166**	32223	507278

[1] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *Proc. of the 26th ICALP*, LNCS 1644, pages 261–270. Springer-Verlag, 1999.

[2] M. Davidson and L. Ilie. Fast data compression with antidictionaries. *Fundam. Inform.*, 64(1–4):119–134, 2005.

[3] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

1068-0314/08 $25.00 © 2008 IEEE
DOI 10.1109/DCC.2008.95