Word-based Statistical Compressors as Natural Language Compression Boosters *

Antonio Fariña[1], Gonzalo Navarro[2], and José R. Paramá[1]
[1] Database Lab, University of A Coruña, A Coruña, Spain.
{fari,parama}@udc.es
[2] Dept. of Computer Science, University of Chile, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract

Semistatic word-based byte-oriented compression codes are known to be attractive alternatives to compress natural language texts. With compression ratios around 30%, they allow direct pattern searching on the compressed text up to 8 times faster than on its uncompressed version.

In this paper we reveal that these compressors have even more benefits. We show that most of the state-of-the-art compressors such as the block-wise bzip2, those from the Ziv-Lempel family, and the predictive ppm-based ones, can benefit from compressing not the original text, but its compressed representation obtained by a word-based byte-oriented statistical compressor.

In particular, our experimental results show that using Dense-Code-based compression as a preprocessing step to classical compressors like bzip2, gzip, or ppmdi, yields several important benefits. For example, the ppm family is known for achieving the best compression ratios. With a Dense coding preprocessing, ppmdi achieves even better compression ratios (the best we know of on natural language) and much faster compression/decompression than ppmdi alone.

Text indexing also profits from our preprocessing step. A compressed self-index achieves much better space and time performance when preceded by a semistatic word-based compression step. We show, for example, that the AF-FMindex coupled with Tagged Huffman coding is an attractive alternative index for natural language texts.

1 Introduction

Traditionally, classical compressors used characters as the symbols to be compressed; that is, they regarded the text as a sequence of characters. Classical Huffman [11] uses a semistatic model to assign shorter codes to more frequent symbols. Unfortunately, the compression obtained when applied to natural language English text is very poor (around 65%). Other well-known compressors are the dictionary-based algorithms of

*Funded in part (for the second author) by Fondecyt (Chile) grant 1-050403, and (for the Spanish group) by MEC (TIN2006-15071-C03-03) and Xunta de Galicia (PGIDIT05-SIN-10502PR and 2006/4).

Ziv and Lempel [15, 16]. These algorithms are usually very fast at compression and especially at decompression (to which they probably owe their popularity), but their compression ratio is still not too good (around 35-40%) on natural language.

It is possible to obtain much better compression by collecting k-th order statistics on the text. These modelers predict the probability of a symbol depending on the context formed by the last k symbols preceding it. This is the case of PPM (Prediction by Partial Matching) compressors [4], which couple such modeling with an arithmetic coder [1]. Compression ratio is very good, close to 22-25%, but they are very slow at compression and decompression. Similar results can be obtained by using a block-wise compressor such as *Seward's bzip2*. It makes use of the Burrows-Wheeler transform (BWT) [7] to obtain a more compressible permutation of the text, and then applies a move-to-front strategy followed by a Huffman coder. In practice, using less memory than ppm-based compressors, bzip2 obtains competitive compression ratios (around 23-26%) and it is much faster at both compression and decompression.

In [12], Moffat proposed to consider the text as a sequence of words instead of characters. By building a word-based model and then applying a Huffman coder, compression ratio was improved from 65% to around 25-30%. Moreover, compression and decompression became much faster. Basically, as words display a more biased distribution of frequencies than that of characters (as Zipf's Law [14] indicates), they become more compressible with a zero-order coder.

Following the word-based approach, in [17] two new byte-oriented compressors were presented. By using bytes instead of bits as the target alphabet (codes are a sequence of bytes rather than bits), compression worsens by around 5 percentage points (30-35%). However, the method becomes much faster at compression and especially decompression. The first technique, called Plain Huffman (PH), is just a Huffman code assigning byte rather than bit sequences to the codes. The second, called Tagged Huffman (TH), reserves the first bit of each byte to mark the beginning of a code and builds the Huffman code over the remaining 7 bits. This leads to a loss of around 3 percentage points in compression ratio, but makes TH a self-synchronizing code, which can be directly searched for a compressed pattern with any technique.

The End Tagged Dense Code (ETDC) [6] retains all the good properties of TH, and obtains better compression ratios. It is a variable-length integer representation of the rank of a word in the frequency-sorted vocabulary. Another recent competitive semistatic proposal was presented in [8].

Text compression has been recently integrated with text indexing, so that one can build an index which takes space proportional to the compressed text, replaces it, and permits fast indexed searching on it [13]. Two practical examples of those so-called "self-index" structures are the Succinct Suffix Array (SSA) and the Alphabet-Friendly FM-index (AF-FMindex) [9][1]. Those indexes work for any type of text.

Overall, the word-based detour has been justified by the interest in achieving fast compression/decompression and direct searching of the compressed text, while maintaining a reasonably competitive compression ratio. In this paper we somehow return to the original goal of optimizing the compression ratio, provided with the tools

[1]Code is available at the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`.

acquired in this detour. The results are surprising, as they show that those compressors designed for natural language actually become *compression boosters* for the best classical methods, both in time and compression ratio. This is the case even with compressors/self-indexes that are supposed to capture the high-order correlations in T, an intriguing fact that we also discuss in the paper.

We show that compressing a text T with ETDC (or any competitive word-based byte-oriented code) is a fast and useful preprocessing step for a general text compressor or self-index. Let us call ETDC(T) the output of this preprocessing. As ETDC(T) reduces $|T|$ to around $|T|/3$, when ETDC(T) is compressed again with a slower compressor X (such as gzip, bzip2 or ppmdi), the whole compression process X(ETDC(T)) is much faster than just X(T). Moreover, X(ETDC(T)) also obtains better compression than X(T). Similarly, we show that a self-index on the preprocessed text is smaller and faster for searching than if applied directly on T[2].

2 ETDC and TH

As explained, TH reserves the first bit of each byte to mark the code beginnings, and uses Huffman coding on the remaining 7 bits to ensure that a prefix code is obtained.

ETDC [6] can be seen as a variation of TH, as it marks the end of a code instead of the beginning. This small change has interesting consequences, as the codes obtained by ETDC are prefix codes independently of the remaining 7 bits, and then Huffman coding is no longer necessary. As a result, ETDC can simply use all the combinations on those 7 bits. That is, the 128 most frequent words in the sorted vocabulary are encoded using the codes from <0000000> to <0111111>. Then, the next 128^2 words are encoded with 2-byte codes from <1000000 0000000> to <1111111 0111111>, and so on. Note that the code associated to a given word does not depend on its frequency, but just on its actual position in the ranked vocabulary.

3 Boosting compression and indexing

The analysis of the byte values obtained by compressing a text T with a byte-oriented word-based compressor (ETDC, PH, etc.) shows that their frequencies are far from uniform. Besides, the same analysis on the output of a bit-oriented encoder[3] displays a rather homogeneous distribution. Figure 1 depicts this situation. This idea led us to consider that the compressed file ETDC(T) was still compressible with a bit-oriented compressor. This could not be a zero-order compressor, because the zero-order entropy (H_0) of ETDC(T) is too high (around 7 *bpc*), and indeed directly using a word-based bit-oriented compressor (like *arith* or that in [12]) achieved better results.

Instead, a deeper study of k-order entropy (H_k) of both T and ETDC(T) exposed some interesting properties of ETDC. The values obtained for H_k for both T and

[2]The price is that we lose the ability of searching for other than whole words and phrases.

[3]Arith, a compressor coupling a word-based modeler with an arithmetic encoder. It is available at `http://www.cs.mu.oz.au/~alistair/arith_coder/`.

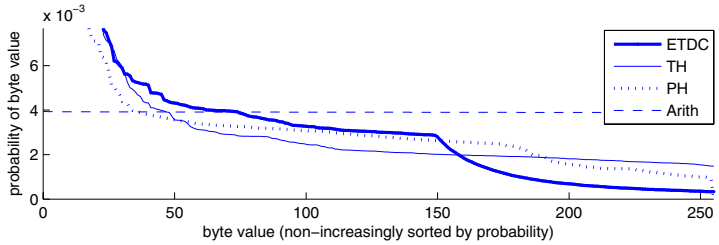


Figure 1: Probability of byte values on TREC-4 Congress Record 93 corpus.

Plain Text						Text compressed with ETDC					
k	H_k	contexts	k	H_k	contexts	k	H_k	contexts	k	H_k	contexts
0	4.888	1	8	0.972	6,345,025	0	7.137	1	8	0.132	12,531,512
1	3.591	96	9	0.837	9,312,075	1	6.190	256	9	0.099	12,854,938
2	2.777	4,197	10	0.711	12,647,531	2	4.642	46,027	10	0.082	13,080,690
3	2.098	51,689	11	0.595	16,133,250	3	2.601	1,853,531	11	0.072	13,252,088
4	1.668	299,677	12	0.493	19,598,218	4	1.190	6,191,411	12	0.061	13,401,719
5	1.430	951,177	13	0.406	22,900,151	5	0.566	9,396,976	13	0.056	13,531,668
6	1.264	2,133,567	33	0.025	43,852,665	6	0.308	11,107,361	49	0.001	14,939,845
7	1.118	3,931,575	50	0.011	46,075,896	7	0.187	12,015,748	50	0.001	14,946,730

Table 1: k-order entropy for plain and compressed CR corpus.

ETDC(T) are given in Table 1 (obtained with software from *PizzaChili*). When regarding the text T as a sequence of characters, the k-order modeler gathers statistics of each character c_i by looking at the k characters that precede c_i. A low-order modeler is usually unable to capture the correlations between consecutive characters in the text, and hence compression is poor (e.g. $H_2 = 4.64$ *bpc*). By switching to higher-order models much better statistics can be obtained [5], but the number of different contexts increases so sharply that compression becomes impractical. For example, existing ppm compressors are able to use k up to 5–10 in practice.

The average length of a word is around 5 bytes in English texts [5], but the variance is relatively high (and raises if we are interested in the distance between two consecutive words). In general, a high-order modeler needs to achieve k near 10 to capture the relationship between two consecutive words.

Modeling ETDC(T) instead of T is clearly advantageous in this aspect. Even though the basic atom is still the byte, the average code length is less than 2 bytes (even considering separators), and the variance is low as codes rarely contain more than 3 bytes (this would require more than $\sum_{i=1}^{3} 128^i = 2,113,664$ different words in T). Hence a k-order modeler can capture the correlations between consecutive words with a much smaller k, or capture longer correlations with a given k.

The argument is not that simple, however, because good compressors like ppmdi do not use a fixed k, but rather administer in the best way they can a given amount of memory to store contexts. Therefore, the correct comparison is between the entropy achieved as a function of the number of contexts necessary to achieve it. For example, around 2 million contexts are necessary to reach H_6 entropy on plain text, which corresponds more or less to a word. About the same is necessary to achieve H_3 on ETDC(T), which also corresponds to a word. The H_k values are not directly

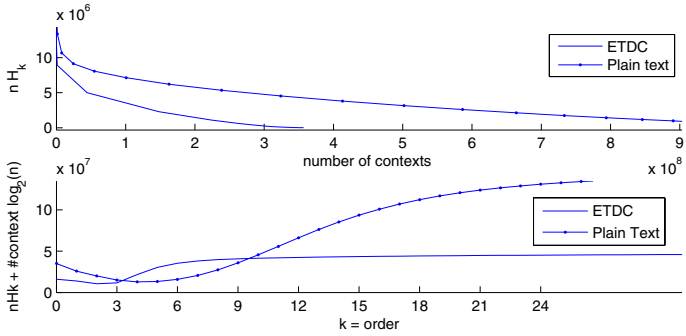


Figure 2: Values nH_k and $nHk + \#contexts \times \log_2(n)$ for plain text and text compressed with ETDC.

comparable because they are in bits per symbol, and ETDC(T) has around one third of the symbols of T. Figure 2 (top) shows the corrected comparison. It displays the value nH_k as a function of the number of contexts, where $n = |T|$.

However, this is not yet considering all the factors. A k-th order compressor pays some price because of the number of contexts. Not only it is usually limited to some amount of memory, but also it has to encode (as a table if it is semistatic, or as escape symbols if it is adaptive) every different context that appears in the sequence. Figure 2 (bottom) gives a more realistic estimation of the size of the compressed text achievable by a k-th order compressor, by penalizing each new context with $\log n$ bits. The minima in the curves show that the compression is expected to be (slightly) better for ETDC(T) than for T, but also that the necessary k is much smaller. This permits faster and less sophisticated modelers to succeed on ETDC(T).

3.1 Using ETDC+X to boost compression

The main idea behind this work is simple. A text is firstly compressed with ETDC, then the resulting data is again compressed with another character-oriented technique X. Therefore, the compressed file is obtained as X(ETDC(T)). Decompression consists also of two steps. The compressed file is firstly decompressed with X^{-1} to obtain ETDC(T). Finally, ETDC decompressor recovers the source text.

Following the guidelines shown in the previous section, we used PPM [4] and BWT [7], two techniques that obtain k-order compression. PPM (from which we chose *ppmdi*) uses the previous k characters of the text as a context, and model the character frequency according to that context. A character c is usually sought at a given k-order model. If the k-order model fails to predict c, an escape symbol is output and a switch to a lower-order model is performed until c is found in a low-order model. Finally, the obtained statistics are used to perform encoding with an arithmetic coder. BWT (from which we chose *bzip2*) obtains a permutation of the text such that characters in the same k-order context are grouped. In such way, the BWT generates a much more compressible text on which a move-to-front coding (MTF) is applied, and this is followed by a zero-order coder (Huffman) encoding.

Compressors from the Lempel-Ziv family [15, 16] are also suitable for this sake.

They obtain compression by detecting sequences of symbols that are repeated as compression progresses, and replacing them by a fixed-size pointer. Lempel-Ziv compression should be useful on top of ETDC, as it would detect repeated phrases in natural language text.

3.2 Using TH+X to boost self-indexing

The same idea of the previous section can be applied to the construction of self-indexes. We chose TH as the base compressor because it generates suffix-free[4] codes. This is a mandatory property when searching for a pattern p, as it permits to compress p and then search for its compressed form directly. We used this idea, joined with AF-FMindex and SSA, to create TH+affm and TH+ssa respectively. As both AF-FMindex and SSA need a terminator for the indexed text (a symbol from the alphabet), we modified TH to ensure that at least 1 byte value does not appear in the compressed text. This loses, in practice, less than 0.05% in compression ratio.

Self-indexes permit counting the number of occurrences of a pattern p in $O(|p|)$ steps. By applying them over TH(T), we search for the encoded words, and hence the counting process is much faster as codewords are shorter. On the other hand, just as high-order compressors, the AF-FMindex produces a structure whose size approaches the k-th order entropy of the sequence, whereas the SSA size is related to the zero-order entropy. We therefore expect the AF-FMindex to be successful in detecting high-order correlations in TH(T), where a smaller k would be sufficient to succeed compared to building on T. This is particularly important because the AF-FMindex is limited in practice to achieve entropies of relatively low k.

4 Experimental results

We used some large text collections from TREC-2: AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF), as well as TREC-4, namely Congressional Record 1993 (CR) and Financial Times 1991 to 1994 (FT91 to FT94). As a small collection we used the Calgary corpus[5]. We also created a larger corpus (ALL) by aggregating them all.

We compared the compressors ETDC, gzip, bzip2, and ppmdi, (setting default options and best compression)[6] against ETDC+gzip, ETDC+bzip2, and ETDC+ppmdi. We also included in the comparison another compressor called MPPM[7] [2] that basically maps words into 2-byte ids, which are later encoded with ppmdi. We provide comparisons in compression ratio, as well as in compression and decompression speed.

An isolated Intel®Pentium®-IV 3.00 GHz, with 4 GB RAM was used in our tests. It ran Debian GNU/Linux (kernel version 2.4.27), using gcc version 3.3.5 with `-O9` optimizations. Time results measure CPU user time.

[4] That is, a code is not a suffix of a longer code.

[5] `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`

[6] `http://rosalia.dc.fi.udc.es/codes`, `http://www.gnu.org`, `http://www.bzip.org`, and `http://pizzachili.dcc.uchile.cl`, respectively.

[7] `http://www.infor.uva.es/~jadiego/`.

	Size				ppmdi		mppm	Cascade ETDC + X			
CORPUS	KB	ETDC	gzip	bzip2	-9	def	-9	gzip	bzip2	ppmdi -6	ppmdi -9
CALGARY	2,081	47.40%	36.95%	28.92%	25.62%	26.39%	26.33%	32.39%	31.67%	28.27%	28.38%
FT91	14,404	35.53%	36.42%	27.06%	23.44%	25.30%	21.90%	26.50%	24.41%	22.86%	22.21%
CR	49,888	31.94%	33.29%	24.14%	20.72%	22.42%	19.14%	22.63%	20.99%	19.72%	18.87%
ZIFF	180,879	33.77%	33.06%	25.11%	21.77%	23.04%	20.56%	23.52%	22.14%	20.69%	20.07%
ALL	1,055,391	33.66%	35.09%	25.98%	22.34%	24.21%	20.83%	24.35%	22.18%	20.90%	19.98%

Table 2: Comparison on compression ratio.

Table 2 shows the compression ratio obtained by ETDC, gzip, bzip2, and ppmdi run over plain text, as well as the compression ratio obtained by ETDC+gzip, ETDC+bzip2, and ETDC+ppmdi. The first two columns in that table indicate the corpus and its size in KB. It can be seen that ETDC+gzip obtains an improvement over gzip of more than 10 percentage points (except in the smallest corpus). These results make ETDC+gzip directly comparable with ppmdi, and overcome the compression ratio obtained by bzip2 in around 1.5 percentage points. ETDC+bzip2 is also able to overcome bzip2 by around 3-4 percentage points and improve the compression ratio of ppmdi (with the exception of the two smallest corpora). ETDC+ppmdi also obtains improvements of around 3-4 percentage points with respect to ppmdi and around 1 percentage point when compared against MPPM. In practice, ETDC+ppmdi works similarly to MPPM. However, while MPPM associates 2-byte ids to each word, which are later encoded with ppmdi, ETDC+ppmdi uses the code associated by ETDC to each word as its id. A very recent work [3] presents a graph-based approach that improves MPPM by 0–4%. Still our results are slightly superior in the cases we are able to compare.

Table 3 shows compression and decompression times. As ETDC is a very fast technique at compression and mainly at decompression, the cascade ETDC+X obtains very good performance. The reason is that bzip2 and ppmdi are very slow. When they are run over the text compressed with ETDC, they have to compress only around 33% of the plain text. As a result, not only ETDC+bzip2 and ETDC+ppmdi compress more than bzip2 and ppmdi respectively, but also they are around 30-40% faster at both compression and decompression (the speedup is not linear because ETDC(T) has more contexts than T, for a given k). Comparing against gzip, ETDC+gzip is also around 25-35% faster at compression (gzip -1 would be faster, but its compression ratio is very poor). Similar results are obtained at decompression (notice that gzip is regarded as a very fast decompression technique, and very popular for that reason).

We note that those measurements have been obtained by just compressing the text with ETDC, and then running the second compressor over the text compressed with ETDC. Better results would be obtained by joining both techniques in such a way that the output of ETDC would be used directly as the input of the second compressor. These would avoid many disk I/Os and performance would be improved.

Figure 3 compares the AF-FMindex and SSA indexes built over text compressed with TH (TH+affm and TH+ssa) against those built over the original text (Plain+affm and Plain+ssa). Using corpus CR, and different values of the parameter *sample-rate* (SR) in the self-indexes (the larger SR, the smaller and slower the index), the figure

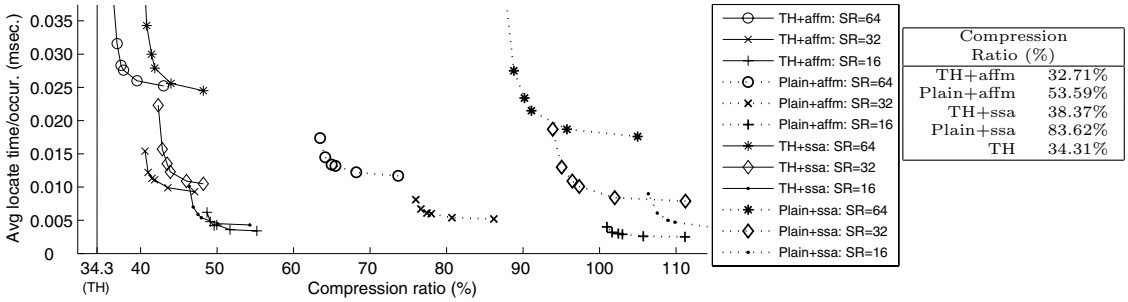
CORPUS	ETDC	gzip	bzip2	ppmdi -9	ppmdi def	mppm -9	Cascade ETDC + X gzip	bzip2	ppmdi -6	ppmdi -9
CALGARY	0.16	0.31	0.77	1.34	1.24	2.30	0.30	0.50	1.12	1.19
FT91	0.98	2.25	4.93	9.02	8.38	14.91	1.64	2.95	6.57	6.82
CR	3.11	7.48	17.03	27.89	26.75	47.89	5.07	9.30	19.86	20.66
ZIFF	11.89	26.29	63.02	105.43	98.81	189.76	20.35	35.19	76.19	80.45
ALL	75.03	157.91	336.81	636.57	599.15	1146.47	118.11	216.17	448.11	470.01

compression time (in seconds).

CORPUS	ETDC	gzip	bzip2	ppmdi -9	ppmdi def	mppm -9	Cascade ETDC + X gzip	bzip2	ppmdi -6	ppmdi -9
CALGARY	0.02	0.04	0.33	1.33	1.31	1.56	0.04	0.20	1.03	1.04
FT91	0.19	0.29	2.15	8.98	9.49	8.91	0.26	1.27	5.98	6.07
CR	0.62	0.95	7.06	27.78	28.15	26.99	0.95	4.02	17.91	18.16
ZIFF	2.25	3.44	25.61	104.53	102.85	105.56	3.59	15.24	68.02	70.57
ALL	14.36	20.70	155.88	630.27	631.09	656.33	21.37	91.43	398.67	408.03

decompression time (in seconds).

Table 3: Comparison of compression and decompression time.



Compression Ratio (%)	
TH+affm	32.71%
Plain+affm	53.59%
TH+ssa	38.37%
Plain+ssa	83.62%
TH	34.31%

Figure 3: Space/time trade-off for AF-FMindex and SSA over compressed and plain text.

shows the space/time trade-off obtained for *locating* all the occurrences of a phrase composed of 4 words. For each SR value we obtain a line rather than a point, by trying out different values for the *rank-factor* parameter. TH+affm and TH+ssa clearly improve the space/time trade-off obtained by their traditional counterparts. The table on the right of Figure 3 shows that, if we reduce the sampling sufficiently (SR=1024 and rank-factor=64, which makes searching slower), the resulting AF-FMindex can become even smaller than applying TH alone, just as k-th order compressors improved upon the result of ETDC(T). On the other hand, the zero-order indexer SSA does not improve upon TH, as expected.

5 Conclusions

We have shown that byte-oriented natural language compressors such as ETDC, TH and PH, are not only attractive because of their acceptable compression ratio and high compression and decompression speed. They can also be seen as a transformation of the text that boosts classical compression/indexing techniques. They transform a text into a much shorter sequence of bytes (around 30-35% of the original text) that is still compressible and indexable. The results obtained by cascading word-based

byte-oriented static compressors with block-wise bzip2, those from the Ziv-Lempel family, and the predictive ppm-based ones, improved their compression ratio as well as their performance in both compression and decompression.

In particular, ETDC+gzip is around 30% faster than gzip at compression, and obtained similar performance at decompression. Moreover, ETDC+gzip improves by around 2 percentage points the compression ratio obtained by bzip2, while it is around 3 times faster at compression and around 7-8 times faster at decompression. Reaching a compression ratio around 24%, ETDC+gzip obtains values that are very close to those of the powerful but slow ppmdi technique. ETDC+gzip is around 5 times faster at compression, and around 25 times faster at decompression. To sum up, ETDC+gzip poses an almost unbeatable trade-off between space and compression and decompression efficiency.

ETDC+bzip2 obtains also a very good compression ratio (around 21-24%) and improves bzip2 by around 3.5-4 percentage points. With respect to performance, bzip2 is overcome by around 75% in both compression and decompression speed.

ETDC+ppmdi becomes actually one of the most effective compressors for natural language text, reaching compression ratios around 20%. It improves the compression ratio of ppmdi by around 3-4 percentage points and that of MPPM by around 0.5-1 percentage points. Again, ETDC+ppmdi is around 30% faster at both compression and decompression than ppmdi alone. With respect to MPPM, ETDC+ppmdi is around 2.5 times faster at compression and improves its decompression speed by around 70%.

Summarizing, we showed that ETDC+gzip, ETDC+bzip2, and ETDC+ppmdi yield an attractive space/efficiency trade-off. ETDC+gzip is very fast and obtains very good compression. ETDC+bzip2 compresses a bit more but is also slower. Finally, ETDC+ppmdi obtains the best compression at the expense of losing some performance.

Indexing also benefits from the use of a compressed version of the text. If we apply the AF-FMindex and SSA over TH compressed text and we set the index parameters in order to obtain a structure of the same size as if we indexed the plain text, we obtain two indexes that are much faster than the traditional ones. If we instead set the parameters to obtain two indexes with the same search speed, the index over the compressed text will occupy 30% less than in the case of the plain text.

For the final version we plan to run more exhaustive experiments, include other natural language compressors and self-indexes, and improve the time performance of our combinations. As future work, we plan to design new byte-codes that improve upon TH while maintaining the properties that make it useful for self-indexing purposes (ETDC was a good replacement for sequential searching, but does not work well for indexed searching). The ideas in [10] could be inspiring for this purpose.

References

[1] N. Abramson. *Information Theory and Coding.* McGraw-Hill, 1963.

[2] J. Adiego and P. de la Fuente. Mapping words into codewords on ppm. In *Proc. 13th SPIRE*, LNCS 4209, pages 181–192, 2006.

[3] J. Adiego, M. Martínez-Prieto, and P. de la Fuente. Edge-guided natural language text compression. In *Proc. 14th SPIRE*, LNCS 4726, pages 14–25, 2007.

[4] T. Bell, J. Cleary, and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communications*, 32(4):396–402, 1984.

[5] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.

[6] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.

[7] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[8] J.S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proc. 12th SPIRE*, LNCS 3772, pages 1–12, 2005.

[9] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM TALG*, 3(2):article 20, 2007.

[10] Sz. Grabowski, G. Navarro, R. Przywarski, A. Salinger, and V. Mäkinen. A simple alphabet-independent FM-index. *International Journal of Foundations of Computer Science (IJFCS)*, 17(6):1365–1384, 2006.

[11] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. of the IRE*, volume 40, pages 1098–1101, 1952.

[12] A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.

[13] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[14] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

[15] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[16] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[17] N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.