Boosting Text Compression with Word-based Statistical Encoding *

Antonio Fariña[1], Gonzalo Navarro[2] and José R. Paramá[1]

[1]Dept. of Computer Science, University of A Coruña, A Coruña, Spain
[2]Dept. of Computer Science, University of Chile, Santiago, Chile
Email: antonio.farina@udc.es; gnavarro@dcc.uchile.cl; jose.parama@udc.es

Semistatic word-based byte-oriented compressors are known to be attractive alternatives to compress natural language texts. With compression ratios around 30-35%, they allow fast direct searching of compressed text. In this article we reveal that these compressors have even more benefits. We show that most of the state-of-the-art compressors benefit from compressing not the original text, but the compressed representation obtained by a word-based byte-oriented statistical compressor. For example, p7zip with a dense-coding preprocessing achieves even better compression ratios and much faster compression than p7zip alone. We reach compression ratios below 17% in typical large English texts, which was obtained only by the slow PPM compressors. Furthermore, searches perform much faster if the final compressor operates over word-based compressed text. We show that typical self-indexes also profit from our preprocessing step. They achieve much better space and time performance when indexing is preceded by a compression step. Apart from using the well-known Tagged Huffman code, we present a new suffix-free Dense-Code-based compressor that compresses slightly better. We also show how some self-indexes can handle non-suffix-free codes. As a result, the compressed/indexed text requires around 35% of the space of the original text and allows indexed searches for both words and phrases.

* A preliminary partial version of this work appeared in [1].

1. INTRODUCTION

1.1. Classic compression

Traditionally, classical compressors used characters as the symbols to be compressed, that is, they regarded the text as a sequence of characters. Classical Huffman [2] uses a semistatic model to assign shorter codes to more frequent symbols. Unfortunately, the compression obtained when applying it to typical English natural language text[3] is poor (around 65%[4]). The dictionary-based algorithms of the Ziv-Lempel family [3, 4], which replace text substrings by previous occurrences thereof, are other well-known compressors. These algorithms are usually fast at compression and especially at decompression, but their compression ratio is still not that good (around 35-40%). The LZMA algorithm is a sophisticated member of this family that is included in the p7zip software.[5] With an augmented dictionary of up to 4 Gigabytes, it achieves attractive compression ratios (around 22-30%), at the cost of slower compression and decompression.

It is possible to obtain better compression by collecting k-th order statistics on the text. This is done by modelers that 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 [5], which couple such modeling with an arithmetic coder [6, 7, 8]. Compression ratio is very good, for example around 17-26% using Shkarin's PPMd[6] but they are very slow at compression and decompression and require much memory. Good results can also be obtained by using a block-wise compressor such as Seward's bzip2.[7] This makes use of the Burrows-Wheeler transform (BWT) [9] to obtain a more compressible permutation of the text,

---

[3]Unless otherwise specified, our measures consider this type of source text.

[4]The size of the compressed text as a percentage of its original size, assuming each original character is encoded in one byte.

[5]http://www.7-zip.org

[6]We use PPMd v.j1 (PPMdj) from http://www.compression.ru/ds

[7]http://www.bzip.org

and then applies a Move-To-Front strategy followed by a Huffman coder. In practice, using less memory than the PPM-based compressors, bzip2 obtains competitive compression ratios (around 24-29%) and it is much faster at both compression and decompression.

Other techniques, such as the *offline* compressors, are particularly well-suited to compress static databases (for example, those that could be stored in CDs or DVDs). Offline compressors make several passes over the text and might use large amounts of time or space at compression. Yet, they should be both fast and memory-efficient at decompression. One compressor of this family is Re-pair [10], which successively replaces the most frequent pair of adjacent source symbols by a new symbol until all the pairs occur only once. It achieves very appealing compression ratios (around 20-31%), yet its compression time or space requirements (depending on the implementation) are rather high.

1.2. Word-based text compression

In [11], Moffat presented a compressor using a zero-order word-based modeler [12] which, combined with a Huffman coder, achieved compression ratios around 25%. With respect to using a character-based modeler, in addition to these improved compression ratios, the election of words as the source alphabet also yielded better compression and decompression times. Basically, as words display a more biased distribution of frequencies than characters (as Zipf's Law [13] indicates), they become more compressible with a compressor building on a zero-order modeler. Another way to see this is that, by using words, one captures k-th order statistics for a reasonable value of k, while ensuring that the model is not too large (as the vocabulary grows sublinearly with the size of the text collection [14]).

Following the word-based approach, two new byte-oriented compressors were presented later [15]. By using bytes instead of bits as the target alphabet (codewords are sequences of bytes rather than bits), compression worsens by around 5 percentage points (to around 30%). However, the method becomes much faster at compression and especially at decompression. The first technique, called Plain Huffman (PH), is just a Huffman code assigning byte rather than bit sequences to the codes, that is, this is just the d-ary Huffam code that appears in Huffman's original paper [2], with d=256. Although it is possible to perform direct searches over PH compressed text [15, 16], these are slower than searches over the second encoding, called Tagged Huffman (TH). TH reserves the first bit of each byte in a codeword to mark the beginning of the codeword 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 *fast self-synchronizing code*,[8] which can be directly

searched for a compressed pattern with any string matching algorithm. While most Huffman codes have the property of *self-synchronization* [17], this may require processing an arbitrary number of codewords. Some compressors introduce synchronization strings or marks to force the resynchronization after a given number of bits/bytes [18, 19]. There is a trade-off between the redundancy introduced by synchronizing strings or marks versus the resynchronization delay. TH has a mark at each byte, which guarantees fast synchronization and allows the use of classical pattern matching algorithms.

Later, the dense codes [20] improved the performance of TH in most aspects. Dense codes are simpler and faster to build than Huffman codes. In addition, they permit the same fast direct searchability and random-decompression capabilities of TH, yet with better compression ratios. The simplest compressor using a dense coding scheme is *End-Tagged Dense Code (ETDC)*, which achieves compression ratios around 31% using a semistatic zero-order word-based model. An improved variant, (s, c)-*Dense Code (SCDC)*, reaches less than 0.3 percentage points over PH compression ratio. Subsequently, other competitive codes were presented [21, 22], but they do not obtain the same search performance as the dense codes.

Although TH and the dense codes traditionally would only allow searching for words or phrases, even allowing errors or patterns with wildcards [15], a recent work [23] shows that it is possible to search not only for words or phrases, but for arbitrary patterns. Yet, searches will perform much slower as searches for a given substring translate into searches for several patterns in parallel.

1.3. Self-indexes

Text compression has been recently integrated with text indexing, so that one can build an index that takes space proportional to the compressed text, replaces it, and permits fast indexed searching on it [24]. Practical examples of those so-called "self-index" structures are the Compressed Suffix Array (CSA) [25], the Succinct Suffix Array (SSA) [26], the Alphabet-Friendly FM-index (AFFM) [27], and the Lz-index (LZI) [28, 29].[9]

Those indexes work for any type of text, achieve compression ratios of 40-60% in typical English texts, and can *extract* any text substring and *locate* the occurrence positions of a pattern string in a time that depends on the pattern length and the output size, but not on the text size (that is, the search process is not sequential). Most of them can also *count* the number of occurrences of a pattern string, much faster than by locating them.

Some word-based self-indexes have been developed,

---

[8]That is, it is possible to detect quickly the beginning of

the codeword containing any position of the compressed stream, hence allowing random decompression from any point.

[9]Their code is available at the *PizzaChili* site, http://pizzachili.dcc.uchile.cl.

such as the WCSA [30] and the WTBC [31]. These achieve compression ratios of 35-40% and provide indexed word-based searches (yet they cannot search for arbitrary text substrings).

1.4. Our results

In text compression, the word-based detour has been justified by the interest in achieving fast compression/decompression and direct searching on the compressed text, while maintaining a reasonably competitive compression ratio. Yet, in this article we use it in a different way. We show that those compressors designed for natural language actually become *compression boosters* for the best classical methods, both in compression time and compression ratio. In addition, they also boost the time of sequentially searching for patterns in the compressed text. They even boost the performance of compressed self-indexes, both in compression ratio and search performance. The fact that we boost the compression ratio of compressors that are already supposed to capture the high-order correlations in T is an intriguing issue that we also discuss in the paper.

We show that compressing a typical English text T with dense codes (or any competitive word-based byte-oriented code) is a fast and useful preprocessing step for a general-purpose text compressor or self-index. Let us call Dense(T) the output of this preprocessing. As Dense(T) reduces $|T|$ to around $|T|/3$, when Dense(T) is compressed again with a backend compressor X (i.e., the final, general-purpose compressor such as gzip, bzip2, p7zip, or PPMdj), in our tests[10] the cascade of compression processes X(Dense(T)) compresses much faster (up to 5 times) than just X(T), in some cases decompresses faster (up to 85%), obtains better compression ratios (up to 11 percentage points less than X(T)), and allows searching for patterns 2-2.5 times faster. In some cases we achieve for the first time faster searching on the compressed text than on the plain text T. Our best compression results, achieved by using PPMdj as the backend compressor, compress the test file Congressional Record 1993 (CR) from TREC-4 to 17%, which is much better than most of the state of the art, and obtain compression and decompression times around 7-85% better than PPMdj alone.

Typical self-indexes are designed to index/compress a sequence of characters. Therefore, since word-based byte-oriented statistical compressors use bytes as their target alphabet, the resulting compressed text can be indexed by the character-based self-indexes. If we want to search for a pattern using a self-index built from text compressed with, for example, TH, we just encode the pattern with the code used to obtain TH(T) and use the index to search for the encoded pattern. As in the case of TH and dense-coded compressed text, it is possible to inspect the vocabulary of words to search for all the

---

[10]See Section 7 for a complete description of such text files.

words (or sequences of words) that include a pattern that is an arbitrary sequence of characters.

If we want to display a snippet of T, we proceed in the reverse way. That is, we extract firstly a portion of the text compressed with TH and finally decompress it. Yet, for this search to be feasible, the code used to preprocess the text should be not only prefix-free (as required for virtually any encoder) but also suffix-free, otherwise the self-index might report false matches. A suffix-free code does not produce codewords that are suffixes of other codewords. TH has this property, but this is not the case of ETDC and SCDC. Another contribution of this article is the development of a suffix-free version of the dense codes, that is called *(s,c,b,o)-Dense Code* (SCBDC). We also show how some self-indexes can be adapted to work correctly over the non-suffix-free SCDC code.

In our experiments, we will show that if we compare the self-indexes built over the uncompressed text T and those built over TH(T), SCBDC(T), or the adapted SCDC(T), and we set the parameters of the self-indexes in order to obtain a structure of the same speed, those built over the word-based byte-oriented codes require between half and one third of the space needed by the ones built over plain text. On the other hand, if we set the parameters to have structures of the same size, the self-indexes build over compressed text are much (up to 130 times) faster. With respect to WTBC and WCSA, published after the preliminary version of this article, these indexes are much more involved and complex to implement than the simple solution presented in this work. Still, the character-based indexes over compressed text get close to the results of WTBC and WCSA, when they do not overcome them.

The paper is structured as follows. In Section 2, we discuss some related work, and some previous concepts are given in Section 3. In Section 4, we analyze why the cascade improves both the compression and the performance of the backend compressors. Section 5 is devoted to the cascade of compressors we test in our experiments, whereas Section 6 discusses the application of the preprocessing step to index text. Section 7 gives the experimental results. Finally, our conclusions are presented in Section 8.

2. RELATED WORK

2.1. Preprocessing for compression

Preprocessing natural-language texts to improve their compression has a long tradition. A popular approach is to parse the source text into syntactic units, such as q-grams, syllables, or words, and run a compressor tailored to handle a sequence of unit identifiers [12, 11, 32, 33, 34, 15, 35, 36, 20].

Alternatively, there are various ad-hoc heuristics [37, 38, 39, 40, 41, 42] that enrich a general-purpose compressor by a preprocessing that chooses some words or other types of syntactic units and replaces them by

different kinds of identifiers, usually assigning shorter identifiers to more frequent units. Those are coupled with a number of very specific tricks that slightly improve the compression ratio.

The best compression results in this line, as far as we know, is the MPPM family. The MPPM adaptive compressor [43] maps text words into 2-byte identifiers that are later encoded with Shakarin/Cheney's PPMdi [44]. Since 2 bytes only permit 2^{16} identifiers, MPPM uses a vocabulary pruning strategy to reuse codewords. A semi-static MPPM2 [45] improves the character-based PPM to reach up to 21% compression ratio.

The technique we present in this article fits in the general approach of parsing the text into syntactic units (words, in our case) to enhance a general-purpose compressor. However, it is more principled. Instead of using simple identifiers, or ad-hoc heuristics to choose some "good" words to encode in an ad-hoc way, we simply use a statistical semi-static byte-oriented compressor on the words. In Section 4 we study the impact of this decision, and later show that the results we obtain with this clean and elegant method are slightly better than those of the MPPM family, the best exponent of the heuristic methods. In addition, our technique has several other advantages in terms of time and memory usage at compression/decompression, and searchability.

2.2. Preprocessing for self-indexing

Apart from the interest of preprocessing the source text to improve the compression, there are also works where such preprocessing has been used to improve indexing. Grabowski et al. [46] built an FM-index over a bit-oriented Huffman compressed version of the text (where characters are still the source symbols). The main problem of this technique is that bit-oriented Huffman does not provide self-synchronization, and thus much additional space is needed to mark the codeword boundaries. In order to solve this issue, they used k-bit symbols as the target alphabet and also presented the FM-KZ variant, where Huffman is replaced by Kautz-Zeckendorf coding, which provides self-synchronization. Their resulting self-indexes obtained better *extract* times than previous structures such as the original FM-index and the SSA, but a worse space/time trade-off for operation *locate*. Typical English texts were compressed to around 90% of their size, at best.

Recently, some word-based self-indexes for natural language text have been developed. The Word-based CSA (WCSA) [30] builds a self-index over a sequence of words rather than characters, using Sadakane's CSA [25], which handles large alphabets well. The WCSA achieves compression ratios of 35-40% and very fast indexed search for both words and phrases.

The Byte-Oriented-Codes Wavelet Tree (WTBC) [31] represents the sequence of words with a clever reordering of the bytes resulting from a word-based byte-oriented encoding of the text (such as PH or SCDC). The reordered representation manages to simulate inverted-index-like searches, that is, the occurrences of single-word queries can be efficiently found, but phrases require an expensive intersection of the ocurrences of their words. Compression is around 35% (some space overhead over PH is necessary for the reordering to work).

Compared to Grabowski et al.'s self-index [46], our proposal uses words as the source symbols, which drastically improves the compression ratios to around 31%. Compared to the WCSA, we use variable-length codes instead of fixed integer values to represent the words. This explains our better compression ratios, yet the WCSA is faster as a function of the pattern length because it regards each word as a symbol while we have to process each of the bytes of its codeword. Compared to the WTBC [31], our data organization allows us to carry out phrase searches as efficiently as word searches, where the WTBC is much slower.

The original version of this article [1] was the first in improving the performance of self-indexes on natural language; the others [30, 31] came (shortly) after and are significantly more complex to implement. Finally, in parallel to our original work [1], Ferragina [47] discussed from a conceptual point of view the WFM-index, which builds an FM-index over text compressed with TH. However, no experimental results were given and, to the best of our knowledge, the WFM-index was not further pursued. Some combinations we present in our experimental results are indeed an implementation of an FM-index after a TH preprocessing. As explained, our new SCBDC improves upon TH.

3. BASIC CONCEPTS

3.1. Empirical entropy

Statistical compression is obtained by combining a *modeler* and a *coder* [48]. The former gathers statistics (number of occurrences of source symbols) of the text, and the second assigns codewords to those source symbols (shorter codewords to more frequent symbols). The empirical entropy of the text lower bounds the performance achievable by some kinds of statistical compressors.

The zero-order empirical entropy of a text T over an alphabet Σ is defined as follows:

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c},$$

where $n = |T|$, n_c is the number of occurrences of character c, and we assume $0 \log_2 0 = 0$. Then $nH_0(T)$ is a lower bound on the output size of any statistical compressor that counts isolated character frequencies in T, independently of their surrounding characters. Huffman coding, when applied over the frequencies n_c, obtains an average codeword length of less than one

target symbol more than the zero-order entropy. When T is seen as a sequence of characters, its zero-order entropy is around 5 bits per character (bpc) (62.5% compression ratio) on natural language, but when seen as a sequence of words it drops to around 2 bpc (25% compression ratio).

Higher-order modelers consider character frequencies depending on the k characters that precede them. Their performance is lower bounded by the k-th order empirical entropy of T:

$$H_k(T) = \sum_{S \in \Sigma^k} \frac{|T_S|}{n} H_0(T_S),$$

where T_S is the string formed by the characters preceded by context S in T. A compressor family approaching the performance of H_k is PPM. Achieving $H_k(T)$ for large k values is challenging because much memory is needed to handle all the $O(|\Sigma|^k)$ models. For the k values that can be reached, the entropy is around $H_k(T) = 1.2$ bpc (15% compression ratio).

3.2. Byte codes

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

ETDC can be seen as a variation of TH, as its encoding scheme marks the end of a codeword instead of the beginning. This small change has interesting consequences, as the codewords obtained by ETDC are prefix-free independently of the remaining 7 bits. Therefore, Huffman coding is no longer necessary, and ETDC can use all the combinations on those 7 bits.

That is, the 128 most frequent words in the vocabulary are encoded using the codewords from $\langle \underline{0}0000000 \rangle$ to $\langle \underline{0}1111111 \rangle$. The next 128^2 words are encoded with 2-byte codewords from $\langle \underline{1}000000\ \underline{0}0000000 \rangle$ to $\langle \underline{1}111111\ \underline{0}1111111 \rangle$ and so on.

The origins of the coding scheme used in ETDC are unclear. We can trace references to similar methods in the literature under the name of byte-codes (bc) or variable-byte coding (Vbyte) [49, 21], and there are possibly earlier ones.

A more sophisticated variant that achieves better compression is SCDC, where the number of byte-values that either mark the last (stoppers) or the other bytes (continuers) of the codeword is not set to 128 but to a tunable value between 1 and 255. Therefore, s values are used as stoppers and $c = 256 - s$ values as continuers.

That is, the s most frequent words in the vocabulary are encoded using the codes from $\langle 0 \rangle$ to $\langle s-1 \rangle$. Then, the next sc words are encoded with 2-byte codes from $\langle s \rangle \langle 0 \rangle$ to $\langle 255 \rangle \langle s-1 \rangle$, and so on. In [20, 50] a discussion on how to obtain the s and c values that minimize the size of the compressed text for a specific word frequency distribution can be found.

4. ANALYSIS OF THE PREPROCESSING USING DENSE CODES

The analysis of the byte values obtained by compressing a text T with a byte-oriented word-based compressor (ETDC, PH, etc.) shows that the frequencies of the byte values generated as the output stream are far from uniform. Instead, the same analysis on the output of an arithmetic coder[11] displays a rather homogeneous distribution. Figure 1 depicts this situation on corpus CR (described in Section 7). This suggests that the compressed file ETDC(T) is still compressible with a bit-oriented compressor. This could not be a compressor based on zero-order modeling, because the zero-order entropy (H_0) of ETDC(T) is too high (around 7 bpc, that is, 87.5% compression ratio), and indeed directly using a word-based bit-oriented compressor (like *arith* or that in [11]) achieves better results.

Instead, a deeper study of k-order entropy (H_k) exposes some interesting properties of ETDC. Using software from *PizzaChili*, Table 1 shows the values of H_k obtained for T, ETDC(T), and a tokenized text where words were considered as the source alphabet (w(T)). The second and third columns show, respectively, H_k and the corresponding compression ratio[12] (with respect to the size of the original file). The fourth column displays the number of generated contexts when regarding the text T as a sequence of characters, that is, the k-order modeler gathers statistics of each character c_i by looking at the k characters that precede c_i. The fifth, sixth, and seventh columns do the same considering the individual byte values of ETDC(T). For the computation of the values of the last three columns, first the text was parsed considering words as the tokens. Then, all the occurrences of each word were substituted by an identifier (integer) assigned to that word. Finally, a modified version of the software from *PizzaChili* that processes integers instead of characters was run over that sequence. The ninth column gives the lower bound of the compression that can be achieved by using the k-th order compression over w(T): it displays $(((\#Words \cdot H_k)/8) \cdot 100)/|T|$, where $\#Words$ is the total number of words in the file, H_k is the corresponding value of the eighth column (in bits per word), and $|T|$ is the size of the file in bytes.

During both the first pass needed to obtain ETDC(T) and the computation of w(T), we used the spaceless word model [51]; that is, if a word is followed by a space, we just consider the word, otherwise both the word and the separator are considered.

When considering plain text, a low-order modeler is usually unable to capture the correlations between

---

[11]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/.

[12]The compression ratios displayed in Table 1 do not include the size of the model; this will be considered soon.
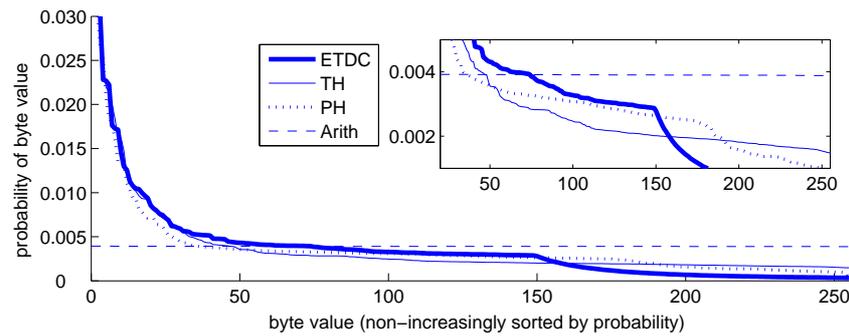
FIGURE 1. Probability of byte values on CR corpus. In the right upper part, there is a magnified area of the values between the byte values 45 and 250.

$k^{th} order$	Plain Text			Text compressed with ETDC			Word tokenized text		
	H_k	c. ratio	#contexts	H_k	c. ratio	#contexts	H_k	c. ratio	# contexts
0	4.888	61.10%	1	7.137	26.85%	1	10.362	25.63%	1
1	3.591	44.89%	96	6.190	23.29%	256	6.105	15.10%	117,714
2	2.777	34.71%	4,197	4.642	17.46%	46,027	3.217	7.96%	1,609,790
3	2.098	26.22%	51,689	2.601	9.78%	1,853,531	1.330	3.28%	4,512,764
4	1.668	20.85%	299,677	1.190	4.48%	6,191,411	0.564	1.40%	6,611,528
5	1.430	17.87%	951,177	0.566	2.13%	9,396,976	0.306	0.76%	7,591,247
6	1.264	15.80%	2,133,567	0.308	1.16%	11,107,361	0.191	0.48%	8,036,769
7	1.118	13.97%	3,931,575	0.187	0.70%	12,015,748	0.145	0.36%	8,280,021
8	0.972	12.15%	6,345,025	0.132	0.50%	12,531,512	0.111	0.28%	8,443,983
9	0.837	10.46%	9,312,075	0.099	0.37%	12,854,938	0.100	0.25%	8,565,274
10	0.711	8.89%	12,647,531	0.082	0.31%	13,080,690	0.084	0.21%	8,669,849
50	0.011	0.14%	46,075,896	0.001	0.01%	14,946,730	0,001	0.004%	9,896,104

TABLE 1. k-order entropy using the CR corpus of 48.7 MB. The H_k values are relative to each text, and thus not comparable to each other. Compression ratios are comparable, but they do not include the size of the model. The impact of the latter can be estimated considering the number of contexts generated.

consecutive characters in the text, thus achieving poor compression, whereas a higher-order modeler needs to handle many contexts. The average length of a word is around 5 bytes in English texts [48], 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 codeword length is less than 2 bytes (even considering separators), and the variance is low as codewords 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.

ETDC(T), using the spaceless model, needs 4 bytes on average (H_3) to capture the correlation between two words. In plain text, H_{10} is 0.711 bytes, whereas H_3 for ETDC(T) is 2.601 bytes, but since ETDC(T) occupies around 35% of T, the resulting value ($2.601 \cdot 0.35 = 0.913$ bytes) fits in the same magnitude of compression of plain text.

This also explains why using, say, ETDC is better than using fixed-length integer identifiers for the words:

As ETDC compresses T better than such a simple identifier encoding, it encodes more words on an average context of length k, and thus the modeler captures more correlations when looking at the last k bytes of the compressed/encoded text.

The argument is not that simple, however, because good compressors like PPMdj do not use a fixed k, but rather administer a given amount of memory to store contexts in the best way they can. Therefore, the correct comparison considers the entropy achieved as a function of the number of contexts necessary to achieve it. As seen, the H_k values in Table 1 are not directly comparable because they are in bits per symbol, and ETDC(T) has around one third of the symbols of T. Figure 2(a) shows the corrected comparison. It displays the value nH_k as a function of the number of contexts, where $n = |T|$ for plain text, $n=|ETDC(T)|$ for ETDC, and $n=\#Words$ for w(T). Thus we show the estimated final output size (model excluded) as a function of the main memory required by the compressor. As seen, w(T) performs better than ETDC(T); this might suggest that the improvements achieved by using ETDC as a preprocessing could be due to the initial word-based parsing of the text applied by ETDC. Still, the memory consumption obtained by ETDC(T) is much closer to that achieved with w(T) than in the case of compressing directly T.

However, this is not considering all the aspects yet. A k-th order compressor pays a price not only in terms of memory required to compress, but also in terms of compression ratio, for the number of contexts it uses. 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(b) gives a more realistic estimation of the size of the compressed text achievable by a k-th order compressor, by penalizing each entry of each context with $\log_2 |\Sigma|$ bits, being $|\Sigma|$ the alphabet size. The results show that although $w(T)$ performs better with a zero-order modeler, with higher orders the cost of storing the contexts rapidly spoils its compression. Moreover, when considering words as source symbols, in the CR corpus $|\Sigma|$ is 117,714, instead of the 256 possible byte values when considering $\text{ETDC}(T)$ as a sequence of bytes. Therefore, the cost of storing each context is much higher in the case of $w(T)$, resulting in a worse compression as the order increases. So the optimum is achieved with a context of $k = 1$ preceding words.

With respect to the normal text, the minima in the curves show that the compression is expected to be (slightly) better for $\text{ETDC}(T)$ than for T, but also that the necessary k is much smaller. For $\text{ETDC}(T)$ the minimum is achieved with $k = 2$, whereas T achieves the best compression ratio with $k = 5$. These values are expected since, as explained, the average codeword length in $\text{ETDC}(T)$ is less than 2 bytes and the average English word is 5 characters long. This permits faster and less sophisticated modelers to succeed on $\text{ETDC}(T)$.

5. BOOSTING COMPRESSION

The main idea behind this work is simple. A text is firstly compressed with a word-based byte-oriented compressor, and then the resulting data is again compressed with another character-oriented technique X. In this work we have chosen the dense codes (either ETDC or SCDC) to preprocess the text, so the compressed file is obtained as $X(\text{Dense}(T))$. Decompression consists also of two steps. The compressed file is firstly decompressed with X^{-1} to obtain $\text{Dense}(T)$. Then the dense decompressor recovers the source text.

Although PH might seem a better choice for preprocessing the text, since it compresses slightly better than SCDC, PH suffers from lack of synchronization that results in a compressed text where direct searches cannot be performed efficiently. We will show that, as dense codes permit us to efficiently perform direct searches, the process of searching the final compressed text with, for example, Lempel-Ziv compression, is much faster than directly searching over $\text{gzip}(T)$. The reason is that the backend compressor has to decompress much less text, which is not in turn decompressed from the ETDC or SCDC format, but directly searched in compressed

| Preprocessing | Backend compressor | | | | None |
	gzip	bzip2	PPMdj	p7zip	
PH	22.27%	20.98%	16.81%	18.53%	31.06%
SCDC	22.43%	20.95%	16.73%	18.48%	31.29%
ETDC	22.63%	20.99%	16.76%	18.49%	31.94%
TH	23.98%	20.72%	16.65%	18.85%	34.28%
None	26.44%	21.64%	16.98%	19.79%	61.23%

TABLE 2. Comparison between byte-oriented codes and a fixed-length compressor with and without a backend compressor, on corpus CR.

form. Section 7.3 discusses the details.

Following the guidelines given in the previous section, we used both a PPM-based [5] and a BWT-based [9] technique as backend compressors. These are techniques that obtain k-th order compression. From the PPM family we chose Shakarin's PPMd v.j1 (PPMdj). It uses the previous k characters of the text as a context, and models 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. Finally, the obtained statistics are used to encode symbols with an arithmetic coder. As a BWT-based technique we used bzip2. The BWT obtains a permutation of the text such that characters in the same k-th order context are grouped. Then a simple local compression of the permuted text achieves k-th order compression of the original text.

Compressors from the Lempel-Ziv family [3, 4] such as the gzip software or the improved LZMA-based p7zip are also suitable for this sake, as they converge to the k-th order entropy. Lempel-Ziv compression should be useful on top of $\text{Dense}(T)$, as it would detect repeated phrases in natural language text.

Table 2 gives some preliminary experiments comparing PH, ETDC, SCDC, and TH as preprocessors for gzip, bzip2, PPMdj, and p7zip. Surprisingly the preprocessing with PH obtained almost the worst values and TH obtained the best results with compressors that obtain k-th order compression (bzip2 and PPMdj). However, apart from their simplicity, we decided to use ETDC and SCDC as they obtain the best overall results (see gzip and p7zip) and the differences are not significant. These experiments were run over corpus CR.

We also included in the experiment a word-tokenized text, where each word is substituted by a fixed-length codeword. Its values are presented in the row labeled *None*. It uses the minimum number of bytes that are necessary to encode all the different words in the text. In the experiment, we used the corpus CR, which contains 117,713 different words, therefore 3 bytes are enough to encode all the words. We also added a column *None* that shows the compression ratio achieved by the byte codes without post-processing. The cell with row *None* and column *None* displays the compression
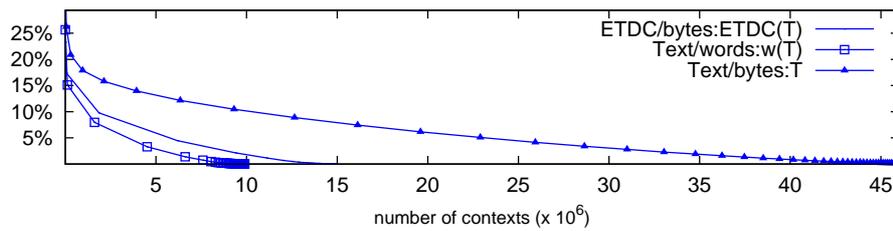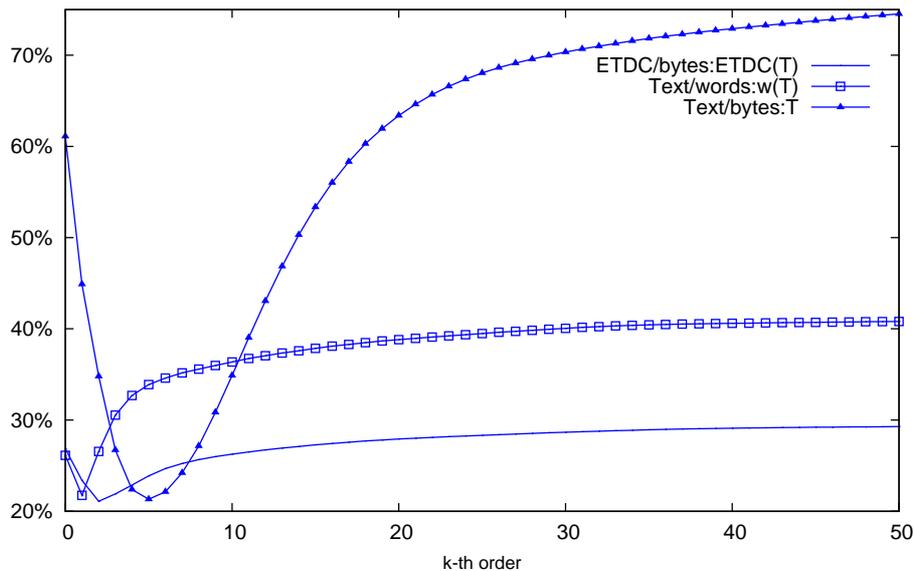
(a) $((nH_k)/|T|) \cdot 100$

(b) $((nH_k + \#contexts \log_2 |\Sigma|)/|T|) \cdot 100$

FIGURE 2. Estimation of the compression ratio for plain text, text compressed with ETDC, and word tokenized text. Figure (a) does not consider the size of the model, whereas Figure (b) includes an estimation of the size of the model.

ratio achieved by the word-tokenized text. As seen, the byte codes obtain better values in all cases, yet PPMdj over word-tokenized text is able to achieve values very close to those of the byte codes. This occurs despite the fact that the average codeword length of the byte codes is less than 2 bytes, whereas the word-tokenized text used fixed 3-byte codewords. Yet, as we set our PPM compressor to capture contexts of lengths up to 12, it seems that the same degree of correlations were captured in both cases.

Although the final differences in compression ratio are not of the same magnitude as before applying the backend compressor, the file size of the word-tokenized text, which is input to the backend compressor, is almost twice that of the file obtained by the byte-code compressors. Thus a byte-code compression preprocessing makes the (comparatively slow) backend compressor/decompressor work over much less data, and therefore boosts compression/decompression times.

We also tried Re-pair [10] as a backend compressor. However, we will see that, in this case, the preprocessing step does not actually improve the results of Re-

pair over the original text. This is because Re-pair naturally starts by forming words via contracting pairs of symbols, and then combines words, so starting from the words makes little difference, as already observed in previous work [52].

6. BOOSTING SELF-INDEXING

The same idea in the previous section can be applied to the construction of self-indexes. Yet, in principle, we cannot use ETDC or SCDC, given that they are not suffix-free codes.[13] This is a mandatory property when searching for a pattern p in a self-index built over compressed data, as it permits to compress p and then search for its compressed form directly. Note that, in online searches over text compressed with a non-suffix-free code, upon a possible match of p, the searcher could still check the previous byte to determine if it was either an actual occurrence of p or a suffix of a longer codeword [20]. However, performing this check on self-indexes

[13]We will see later that some self-indexes can index text compressed with SCDC, but additional modifications in both the indexes and the SCDC are needed.

would turn into linear-time searching, whereas indexed searching should take sublinear time.

For example, assume that we have preprocessed a text with ETDC and then a CSA has been built on it. Assume also that ETDC has assigned the codeword $\langle 26 \rangle$ to word "the" and codeword $\langle 200 \rangle \langle 26 \rangle$ to "journal", and that we want to *count* the occurrences of "the" using the CSA. Therefore, we are interested in counting the occurrences of $\langle 26 \rangle$ preceded by a byte with a value lower than 128 (a stopper in ETDC [20]) to ensure that the occurrences of $\langle 26 \rangle$ reported are actual occurrences of "the" rather than a suffix of a longer codeword such as that of "journal". Unfortunately, the CSA will report all the occurrences of $\langle 26 \rangle$, and we will have to check later if those occurrences are valid or not, one by one. Therefore, being x the number of occurrences of $\langle 26 \rangle$, counting the occurrences of "the" would cost, due to the required check, $\Omega(x)$ time in our CSA built on ETDC compressed data. However, that operation would have costed only $O(m \log n)$ time on a regular CSA, being m the length of the search pattern. Note that, if we use a suffix-free code (such as TH) that check is no longer needed and the counting operation costs only $O(m \log n)$.

As a result, we can still use TH as the base compressor because it generates suffix-free codewords. In addition, we have also developed a new suffix-free member of the family of dense codes which is discussed in the next section.

6.1. SCBDC: $(s, c, b, o)-$dense codes

Observe that the codewords of both ETDC and SCDC are made from two disjoint sets of byte-values; the last byte of each codeword uses a set of byte-values (stoppers) and the rest of bytes use a different set of byte-values (continuers). Since a codeword always ends with a stopper, a shorter codeword cannot be a prefix of a longer codeword. Yet, both the one-byte codewords and the last byte of the longer codewords share the same byte-values. For example, the codeword $\langle 0 \rangle$ is a suffix of the codeword $\langle 128 \rangle \langle 0 \rangle$. Similarly a two-byte codeword like $\langle 128 \rangle \langle 0 \rangle$ is a suffix of the longer codeword $\langle 129 \rangle \langle 128 \rangle \langle 0 \rangle$, and so on.

We designed the $(s, c, b, o)-$dense code (SCBDC), which overcomes the previous problems, and still keeps the prefix-free property of ETDC and SCDC. To achieve a prefix- and suffix-free dense code, instead of using only two different sets of byte values, the new SCBDC has four sets of byte values:

1. *One-byte codewords* have its own set of byte values. This prevents one-byte codewords from being either prefixes or suffixes of longer codewords.
2. *Beginners* are first-byte values of codewords of two or more bytes. Since the first byte value of a codeword cannot occur in any other part of a codeword, we ensure that shorter codewords cannot be suffixes of longer codewords.
3. *Stoppers* are last-byte values of codewords of two or more bytes. These bytes work like in ETDC and SCDC to achieve a prefix-free code. Since the value of the last byte of a codeword cannot occur in any other position of a codeword, shorter codewords cannot be prefixes of longer codewords.
4. *Continuers* are the rest of byte values, which are used in the middle bytes of codewords of length three or more.

In Figure 3, each set of bytes is represented with a rectangle with a different filling pattern. In the upper part of the figure, we can see that both SCDC and SCBDC are prefix-free codes. Observe, for example, the second byte of the codewords of length 2 and 3, in both SCDC and SCBDC. In the case of the codeword of length 2, this byte is a stopper, whereas in the case of the codeword of length 3, this byte is a continuer. Therefore these two bytes are different values for sure, and then the 2-byte codeword cannot be a prefix of the 3-byte codeword. In the bottom part of the figure we can observe that the last byte of SCDC codewords is always from the same set of byte values (stoppers) and the rest of bytes are continuers, therefore shorter codewords can be suffixes of longer codewords. This problem is solved in SCBDC: *i)* one-byte codewords use different values with respect to all the others, and *ii)* the values for the first byte of longer codewords are taken from a different set of values (beginners) thus avoiding the suffix problem.

Both SCDC and SCBDC adapt their encoding schemes to the word frequency distribution of the source words. SCDC computes the s and c values that achieve the best compression, whereas SCBDC has to compute two additional values. As in SCDC, the most practical way to tune its parameters is performing a brute-force search [20] which computes the size of the compressed text for each combination of s, c, b, and o, and finally chooses the best value. By precomputing cumulative frequencies of the vocabulary sorted by decreasing frequency, the whole computation takes time $\binom{2^l}{3} \approx \frac{1}{6} \cdot 8^l$, where l is the number of bits of the symbols. In our case $l = 8$ as we use bytes; this involves less than 3 million accesses to the array of cumulative frequencies. The time consumed by this process harms the compression time of short files, while it becomes more negligible as we compress larger files (see Section 7). For example, in our test computer, the search for s, c, b, and o takes 0.0324 sec in the corpus CR, which represents around 2% of the compression time. In a shorter corpus of 2 Mbytes (see Section 7 for a description of the CALGARY corpus), this time has a considerable impact, given that the whole compression process takes 0.11 sec, which is 37% slower than the compression time of the same file using SCDC.

To decrease the costs of this process we developed a procedure that, instead of computing all the possible combinations, uses a heuristic to check only some of
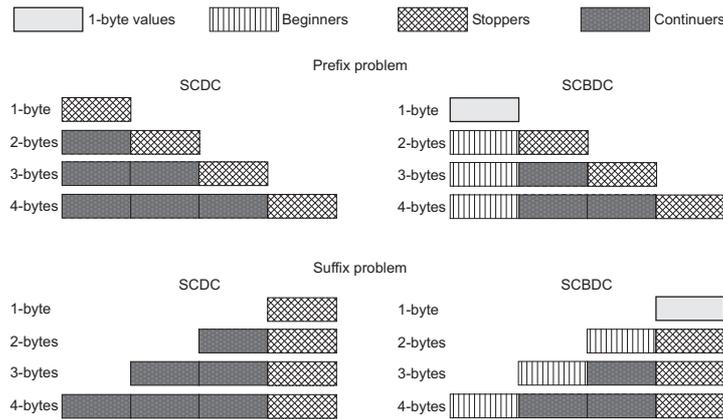
FIGURE 3. Suffixes and prefixes in SCDC and SCBDC.

them. It works as follows:

- Instead of checking all the combinations of s, c, and b for the 253 possible values of o, only 10 tests are executed. Those tests are run for values spaced at intervals of 25, that is, the process checks values $2, 27, 52, \ldots, 253$.
- The previous process gives a first best candidate o_1 value. Then the search is now restricted to the interval $[o_1 - 20, o_1 + 20]$ testing values spaced at intervals of 5.
- The previous step gives a second best candidate o_2 value. Now the search is restricted to the interval $[o_2 - 10, o_2 + 10]$, where all values are tested.

This last pass gives the final o, s, c, and b values. In all our tests, this process gave the same values as the brute-force version. In the corpus CR, this heuristic search takes 0.0045 seconds, that is 7 times faster than the brute-force approach, and in the case of the CALGARY corpus, the whole compression process consumes 0.09 sec, only 6% slower than SCDC. Yet, in the experiments of Section 7, we used the version with the brute-force process.

6.2. Indexing

We applied the same idea used in compression for boosting indexing, but using TH and SCBDC which, as explained, are prefix and suffix-free codes. We first apply the compressor to the original text. This produces a sequence of bytes reduced to around 33% of the original size. The resulting sequence is then self-indexed. In order to search for an occurrence of a pattern p, we encode the pattern (with the same code used to compress the original text) and provide its codeword C_p to the self-index, that directly searches for pattern C_p (just considering it as a sequence of bytes). Note that, since the time needed to search for a pattern p typically depends on its length, and given that it usually holds that $|p| > |C_p|$ we expect to obtain improved search times.
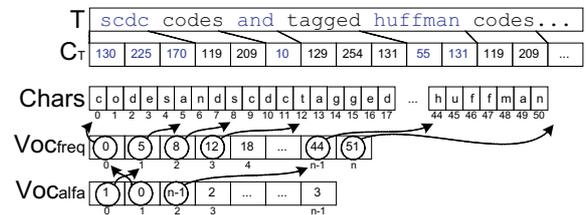

FIGURE 4. In-memory data structures used to hold the vocabulary.

The only additional structure needed is a table to hold the correspondence between original words and the codewords that represent them in the compressed text. To allow searches (encoding the search pattern is needed), this structure has to provide the codeword corresponding to a word. This implies that the words should either be sorted or kept in a hash table. In addition, during decompression the reverse process is needed: decoding a codeword C_i recovers the word at the i-th entry in the vocabulary sorted by frequency. Therefore, keeping a list of words sorted by frequency is also mandatory. Figure 4 shows the data structures that are used to manage the search operations supported by the self-indexes built over text compressed with SCBDC and TH.[14] Note that $Chars$ and Voc_{freq} keep the list of words sorted by frequency: the i-th word is in $Chars[(Voc_{freq}[i]) \ldots (Voc_{freq}[i+1]-1)]$. Note also that Voc_{alpha} contains pointers to Voc_{freq} that permit us to access the words in alphabetical order (so binary searching for a pattern p in the vocabulary is possible). We studied three different variants to handle these vocabulary structures ($Chars$, Voc_{freq}, and Voc_{alpha} arrays), with different space/time trade-offs: *(i)* uncompressed representation; *(ii)* $Chars$ is kept uncompressed, whereas Voc_{freq} and Voc_{alpha} are compacted using $\lceil (\log_2 |Chars|) \rceil$ bits and $\lceil \log_2 n \rceil$

---

[14] Additionally, SCBDC also needs the values of the parameters o, s, c, and b. TH has to keep the shape of the canonical Huffman tree used during the compression.

bits respectively (n represents the number of different words); and *(iii) Chars* is compressed with a char-oriented Huffman and the other structures are handled as in case (ii) (yet pointers to bit positions are necessary in this case). The best space/time trade-off was obtained by the approach (ii).

In our implementation, the addition of the vocabulary structures represents an increase of around 5-8% in the size of the index. Yet, as explained, the index built over compressed text is still much more space efficient than that built over plain text.

We modified both TH and SCBDC to ensure that at least one byte value remains unused in the compressed text. This is generally required by the self-indexes, that will usually choose that value as a terminator during the indexing process. In practice, this modification loses less than 0.05% in compression ratio.

We considered four self-indexes from the *PizzaChili* site: we used the 2.0 version of AFFM, the 3.0 version of SSA, the LZ-Index-4 version [29] of LZI, and the text version of CSA. As in the case of high-order compressors, AFFM, LZI, and CSA produce structures whose sizes approach the k-th order entropy of the indexed sequence (yet in some of them, like the LZI, the constant may be large), whereas the SSA size is related to the zero-order entropy. Therefore, we expect the AFFM, LZI, and CSA to be successful in detecting high-order correlations in TH(T) and SCBDC(T), where a smaller k would be sufficient to succeed compared to those indexes built on T. This is particularly important because the AFFM, LZI, and CSA are limited in practice to achieve entropies of relatively low k.

6.3. Boosting self-indexing without the suffix-free property: SCDC

In the previous sections, we have shown that by using byte-oriented codes having the suffix-free property, any char-based self-index can be directly built on top of the compressed data without the need of modifying its implementation. In such way, we obtain a universal preprocessing step that boosts any self-index.

On the one hand, using SCBDC instead of SCDC yields a little loss in compression, as we will show in Section 7.1. On the other hand, using SCDC would require the index to check whether each occurrence of a pattern is preceded by a stopper (a byte value in the range $[0, s-1]$).

In this section we show that it is still possible to use SCDC as the backend compressor, and build AFFM, CSA, and SSA indexes on the compressed data, so that the expensive check can be avoided. Yet, some modifications are required in both the SCDC and the search procedure of the indexes.

Those self-indexes (let us call them generically SA, for *suffix arrays* [53]) represent all the suffixes of the text they index in lexicographic order, so that they find all the occurrences of a pattern q by identifying

a lexicographic range of suffixes starting with q. The idea is that, although $[0, s-1] \cdot q$ does not lead to a lexicographic range of suffixes, its reverse, $q^{rev} \cdot [0, s-1]$ (where q^{rev} means q read backwards) does.

To use this property, instead of building the self-indexes on top of SCDC(T), we build them over SCDC(T)^{rev}, that is, over the text compressed with SCDC[15] and then read backwards. By doing so, we will have to search not for the compressed representation of a pattern p, that is, SCDC(p), but for its reversed counterpart, SCDC(p)^{rev}, followed by a range of characters $[0, s-1]$.

This kind of search is particularly convenient in suffix array indexes using *backward-search* such as AFFM, CSA, and SSA, as it permits us to perform the check to avoid false matches very efficiently. Backward search processes the characters of the search pattern in backward order, and starting with the range $[0, s-1]$ only requires that the search does not start over the whole array of suffixes, but only on the range of suffixes that begin with a character in the range $[0, s-1]$.

For example, assume that a compressed pattern $cp = \langle 200 \rangle \langle 26 \rangle$ is being searched for on a self-index SA built over text compressed with SCDC. Therefore, the CSA will firstly find intervals over CSA(SCDC(T)) with suffixes starting by $\langle 26 \rangle$, and in the next step it will reduce that range to an interval where suffixes start by $\langle 200 \rangle \langle 26 \rangle$. Therefore, the backward search involves only processing the two bytes of the pattern cp, but in a later step we will have to extract one extra byte before all those occurrences to filter out only the occurrences of cp preceded by a stopper.

However, if we build our CSA on top of SCDC(T)^{rev}, we can directly search for $cp' = \langle 26 \rangle \langle 200 \rangle \langle [0 \ldots s-1] \rangle$. In this case, the first step of the backward search is modified so that it focuses in locating a range containing suffixes that start with a stopper, hence performing the needed check in a cheap way. Next, the backward search continues considering the bytes $\langle 200 \rangle$ and $\langle 26 \rangle$ as usual. In this case, all the occurrences reported are guaranteed to be proper occurrences of cp'. Therefore, the only additional cost of the search is that it consists of $|cp| + 1$ steps rather than of just $|cp|$.

To sum up, we have shown that SCDC can still be used to boost self-indexing. Yet, it requires modifying the implementation of the indexes so that it permits us to avoid reporting false occurrences in an efficient way.

7. EXPERIMENTAL RESULTS

In this section, we used some large text collections from TREC-2 and TREC-4, namely Ziff Data 1989-1990 (ZIFF), Congressional Record 1993 (CR), and Financial Times 1991 (FT91). As a small collection, we used

---

[15]As in SCBDC and TH, we also modified SCDC so that one byte value, (0 in this case) does not appear in any codeword.

	CALG	FT91	CR	ZIFF	ALL
SIZE (MB)	2.00	14.07	48.72	176.74	1030.65
PH	46.24%	34.64%	31.06%	32.88%	32.83%
TH	50.23%	37.93%	34.28%	36.31%	36.38%
ETDC	47.40%	35.53%	31.94%	33.77%	33.66%
SCDC	46.61%	34.89%	31.29%	33.06%	33.02%
SCBDC	49.64%	37.47%	33.93%	35.83%	36.01%

TABLE 3. Comparison on compression ratio of the byte-oriented codes.

	CALG	FT91	CR	ZIFF	ALL
PH	25.09	28.65	30.68	28.68	27.71
TH	25.40	28.59	30.76	28.83	27.87
ETDC	25.73	28.42	30.91	28.81	27.99
SCDC	25.09	28.30	30.76	28.75	27.82
SCBDC	18.31	27.37	30.28	28.94	27.95

TABLE 4. Comparison on compression speed (MB/sec) of the byte-oriented codes. The values refer to MB from the original files processed per second.

the Calgary corpus (CALG).[16] We also created a larger corpora (ALL) by aggregating them all, AP Newswire 1988, and Financial Times 1992 to 1994. Table 3 shows the sizes of these corpora. From now on, any claim will be made with respect to this set of corpora.

An isolated Intel®Xeon®-E5520@2.26GHz with 72 GB DDR3@800MHz RAM was used on our tests. It ran Ubuntu 9.10 (kernel 2.6.31-19-server), using gcc version 4.4.1 with -O9 options. Time results refer to cpu user time.

In all cases, compression times assume that the compressors or the cascades are fed with the text in plain form. In the same way, decompression times include the complete process of recovering the original text. We recall that our compression ratios give the size of the compressed file as a percentage of the original size in plain form (text) and, in this section and subsequent sections, they include the size of any structure needed to recover the original file (model and/or dictionary).

7.1. (s,c,b,o)-Dense Code

We start by analyzing our suffix-free dense code. We compare it with the most popular byte-oriented codes: PH, TH, ETDC, and SCDC. Table 3 shows the compression ratios, whereas Tables 4 and 5 show the compression and decompression speeds, respectively.

Obviously, by having more restrictions on its byte assignments, SCBDC necessarily compresses less than SCDC. Yet, its main competitor is not the SCDC, but the TH, as this code is also suffix-free. In our experiments, SCBDC outperforms TH in compression ratio (by around 1%). Differences in compression and decompression speed are negligible in all cases, and the reasons should be found in small details like

---

	CALG	FT91	CR	ZIFF	ALL
PH	88.36	101.19	105.91	92.77	94.11
TH	84.68	97.01	106.14	95.02	94.90
ETDC	96.78	97.01	108.51	99.35	99.18
SCDC	81.29	96.34	102.57	96.16	96.20
SCBDC	78.17	92.54	100.24	90.82	91.04

TABLE 5. Comparison on decompression speed (MB/sec) of the byte-oriented codes. The values refer to MB of the final file recovered per second.

compiler optimizations or cache misses. The only remarkable differences are that SCBDC is a bit slower at decompression (under 9%) and during the compression of small files, as explained in Section 6.1.

As a member of the dense code family, SCBDC is easier to program than TH, since it does not need to deal with any kind of tree construction process. In addition, as we will see in Section 7.4, the byte ordering of SCBDC is predicted better by CSA and especially by AFFM. TH.

7.2. Boosting compression

In this section, we compare plain compressors with the cascade of first applying a dense compressor and then a backend compressor. Our comparison includes the following compressors:

- ETDC and SCDC standalone.[17]
- The *Re-pair*[18] compressor coupled with a bit-oriented Huffman.[19]
- Gnu *gzip,*[20] a Ziv-Lempel-based compressor.
- The *p7zip* compressor,[21] which is a LZMA compressor with a dictionary of up to 4 Gigabytes. It uses 8 CPUs during compression on our machine.
- Seward's bzip2,[22] a compressor based on the Burrows-Wheeler transform.
- As a representative of the PPM family, we used Shakarin's PPMd v.j1[23] (PPMdj).

We used the maximum compression options whenever they existed, that is, gzip -9, bzip2 -9, and PPMdj -m256 -o12 -r1.[24] All these compressors X were compared with the cascade of ETDC+X and SCDC+X. In what follows, we will use the term "dense+X" to refer to "ETDC+X and SCDC+X".

We also included in the comparison the MPPM[25] compressor using the -9 (maximum compression) option as well. This compressor is not included in the cascade, since it already handles words.

---

[16]We concatenated in a single file the subset of the text files of the Calgary collection: book1-2, bib, news, and paper1-6. The Calgary corpus is available at ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus.

[17]http://vios.dc.fi.udc.es/codes

[18]http://www.cbrc.jp/~rwan/software/restore.html.

[19]http://cs.mu.oz.au/~alistair/mr-coder.

[20]http://www.gnu.org.

[21]http://www.7-zip.org/

[22]http://www.bzip.org.

[23]http://www.compression.ru/ds/

[24]PPMdj allows up to order 16 (-o16), but better results were obtained with the -o12 option.

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

In the compilation of ETDC, SCDC, and MPPM, we used the `-m32` option, as those compressors were designed for 32 bit machines. In the case of PPMdj, we also used `-m32` option, as we could not compile the 64-bit version in our test machine. In the case of Re-pair we had to compile it with `-m64` option, otherwise it cannot compress the ZIFF and ALL corpus. Finally, we used the 64-bit versions of gzip, bzip2, and p7zip distributed for our test system.

Table 6 shows the compression ratios achieved in our tests. The first three lines show the values of ETDC, SCDC, and MPPM applied over plain text. In the rest of lines, the first column indicates if a preprocessing step was applied or not. A *"None"* means that the compressor in the second column was applied over plain text. Otherwise, the first column informs about the compressor (either ETDC or SCDC) used as a preprocessing step, and the second column shows the backend compressor.

It can be seen that dense+gzip obtains an improvement over gzip of more than 10 percentage points (except in the smallest corpus). These results permit dense+gzip to overcome bzip2 by around 1.5 percentage points. Similarly, dense+bzip2 improves bzip2 by around 3-4 percentage points (again excepting the smallest corpus).

Dense+p7zip overcomes p7zip by around 3 percentage points, if we except again the smallest corpora. This cascade overcomes the values of MPPM (in large corpora) and Re-pair, but, as we show below, with much better compression and decompression times.

Dense+PPMdj obtains improvements in the two largest corpora of around 0.5 percentage points with respect to PPMdj, therefore obtaining impressive values of compression ratio of around 17%, the best one of our experiments. In practice, dense+PPMdj works similarly to MPPM. While MPPM associates 2-byte identifiers to each word, which are later encoded with a PPM encoder (PPMdi[26]), dense+PPMdj uses the codeword associated by ETDC or SCDC to each word as its id. We are not including results for dense+PPMdi, since PPMdi is less powerful than PPMdj. However, to have a fairer comparison with MPPM, we also run some limited experiments. We obtained that ETDC+PPMdi overcomes MPPM in compression ratio by around 0.5-1 percentage points.

Our rough estimation in Figure 2(b) predicted that ETDC over k-th order compressed text would achieve at most 21% compression ratio. Yet, Table 6 shows that, in corpus CR, it barely overcomes PPMdj over plain text, but with compression ratios around 16%. This difference is due to the sophisticated PPMdj, which is an improved PPMd version called *PPM with Information Inheritance* [44].

The only compressor that obtains worse compression ratios by applying the cascade is Re-pair, for the reasons

---

[26]Available at the *Pizza-Chili* website.

Boost	Comp	CALG	FT91	CR	ZIFF	ALL
				CORPUS		
ETDC		47.40%	35.53%	31.94%	33.77%	33.66%
SCDC		46.61%	34.89%	31.29%	33.06%	33.02%
MPPM -9		26.33%	21.90%	19.14%	20.56%	20.83%
None	gzip -9	36.84%	36.33%	33.18%	32.98%	35.00%
ETDC		32.30%	26.47%	22.63%	23.50%	24.34%
SCDC		32.20%	26.31%	22.43%	23.29%	24.19%
None	bzip2	28.92%	27.06%	24.14%	25.11%	25.98%
ETDC		31.67%	24.41%	20.99%	22.14%	22.18%
SCDC		31.23%	24.31%	20.95%	22.12%	22.17%
None	p7zip	29.96%	25.52%	21.64%	22.99%	22.80%
ETDC		29.38%	22.74%	18.49%	19.52%	19.25%
SCDC		29.48%	22.75%	18.48%	19.48%	19.21%
None	PPMdj	**25.36%**	**20.31%**	**16.88%**	18.11%	17.87%
ETDC		28.04%	21.07%	**16.76%**	**17.57%**	**17.34%**
SCDC		27.97%	21.03%	**16.73%**	**17.54%**	**17.31%**
None	Re-pair	31.20%	24.00%	20.16%	20.33%	20.00%
ETDC		34.00%	25.12%	20.92%	21.55%	21.19%
SCDC		33.57%	24.67%	20.44%	21.06%	20.77%

TABLE 6. Comparison on compression ratio. We emphasize the best one and those within 1% of it.

already explained.

Table 7 shows compression speed. In compression, on the one hand, the backend compressor is run over a text compressed with ETDC or SCDC, so it has to compress only around 33% of the original text. On the other hand, the cascade needs to apply two procedures instead of just one. However, since ETDC and SCDC are much faster than most of the backend compressors, the benefits surpass the costs and the cascades are up to 5 times faster than the corresponding backend compressor. More precisely, the cascades are around 38-87% faster in the case of gzip, 24-76% with bzip2, between 3 and 5 times when using p7zip, 7-80% with PPMdj, and even in the case of Re-pair, where the cascades performed worse in compression ratio, there is an improvement in compression speed between 24% and 210%.

Table 8 displays the decompression speed. Here, as some backend decompressors are very fast, the cascades do not always improve times. In fact, the cascades are between 20% and 35% slower in the case of gzip, between 24% and 57% in the case of p7zip, and between 7% and 25% in the case of Re-pair. However, when using bzip2 and PPMdj, the cascades obtain a speed-up of around 9-42% and 8-85%, respectively.

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

7.2.1. Discussion
Figure 5 shows trade-offs between compression ratio and compression/decompression speed achieved by the

Boost	Comp	CORPUS				
		CALG	FT91	CR	ZIFF	ALL
ETDC		25.73	28.42	30.91	28.81	27.99
SCDC		25.09	28.30	30.76	28.75	27.82
MPPM -9		1.77	1.90	2.06	1.88	1.80
None	gzip -9	9.24	9.63	9.86	10.83	10.21
ETDC		12.70	15.80	18.11	15.25	16.38
SCDC		12.70	15.98	18.45	15.36	16.60
None	bzip2	6.56	6.42	6.39	6.51	6.34
ETDC		8.47	10.19	11.23	10.74	10.35
SCDC		8.13	10.19	11.23	10.74	10.37
None	p7zip	1.49	1.01	0.95	0.91	0.90
ETDC		4.32	4.82	4.57	3.60	3.68
SCDC		4.42	4.92	4.73	3.70	3.79
None	PPMdj	3.33	2.98	2.41	1.89	1.70
ETDC		3.57	3.97	4.34	3.15	2.89
SCDC		3.44	3.92	4.33	3.16	2.89
None	Re-pair	1.34	1.14	0.88	0.50	0.13
ETDC		2.78	2.38	1.68	0.75	0.16
SCDC		2.82	2.41	1.72	0.78	0.17

TABLE 7. Comparison on compression speed (MB/sec, where the MBs refer to the original file).

Boost	Comp	CORPUS				
		CALG	FT91	CR	ZIFF	ALL
ETDC		96.78	97.01	108.51	99.35	99.18
SCDC		81.29	96.34	102.57	96.16	96.20
MPPM -9		2.57	3.02	3.39	3.15	3.17
None	gzip -9	67.74	82.74	82.57	83.72	81.60
ETDC		50.81	63.94	70.61	64.47	65.03
SCDC		50.81	61.16	68.62	63.31	64.22
None	bzip2	16.94	16.95	17.98	17.61	17.27
ETDC		18.48	22.33	25.51	23.93	23.85
SCDC		18.48	22.33	25.37	23.90	23.76
None	p7zip	29.03	38.02	44.70	41.96	42.55
ETDC		18.48	28.13	35.56	33.65	34.40
SCDC		18.48	28.13	34.80	33.33	33.95
None	PPMdj	3.08	2.80	2.30	1.82	1.63
ETDC		3.33	3.82	4.26	3.11	2.86
SCDC		3.23	3.73	4.24	3.11	2.87
None	Re-pair	50.81	46.89	45.96	42.26	35.52
ETDC		40.65	41.37	42.74	37.50	33.26
SCDC		40.65	42.62	41.64	37.50	33.25

TABLE 8. Comparison on decompression speed (MB/sec, where the MBs refer to the target file).

cascades against compressors applied over the plain text. In order to avoid cluttering the figure, we did not include the ETDC and its cascades, as their values are very close to those of SCDC.

In particular, with respect to gzip, dense+gzip obtains an improvement of 14-48% in compression ratio and 38-87% in compression speed, yet the decompression performance worsens by around 20-35%. Moreover, dense+gzip improves the compression ratio obtained by bzip2 by around 1.5 percentage points, while it is around 2-3 times faster at compression and around 3-4 times faster at decompression. Reaching a compression ratio around 22-24%, dense+gzip obtains values that are very close to those of the powerful but slow p7zip technique. Dense+gzip is from 8.5 to 19 times faster than p7zip at compression and around 53-75% at decompression. Dense+gzip is around 4 percentage points worse than MPPM -9 and Re-pair in compression ratio. However, Dense+gzip is 9-123 times faster than Re-pair and around 7-9 times faster than MPPM at compression, being around 0-83%

faster than Re-pair and around 20 times faster than MPPM at decompression. Finally, dense+gzip loses 6-7 percentage points against the unbeatable PPMdj, but it is around 4-10 times faster at compression and 16-40 times faster at decompression. To sum up, dense+gzip poses an almost unbeatable trade-off between space and compression and decompression efficiency.

Dense+bzip2 obtains also a very good compression ratio (around 22-24% in large corpora) and improves bzip2 by around 3-4 percentage points; these values are on a par with the compression ratios achieved by p7zip. With respect to performance, bzip2 is overcome by around 24-76% at compression and 9-42% at decompression speed.

Dense+p7zip compresses around 3 percentage points more than p7zip in large texts. Dense+p7zip also improves the compression ratio achieved by Re-pair in all the corpora of our experiments. The same occurs with MPPM in the large corpora. Yet, dense+p7zip is between 3-28 times faster at compression when comparing with Re-pair. It is also around twice as fast as MPPM at compression and 7-10 times faster at decompression than MPPM.

With respect to PPMdj alone, dense+PPMdj improves its compression ratios by around 0.5 percentage points in large texts and obtains compression and decompression times around 7-85% better. Dense+PPMdj overcomes by far all the other compressors in compression ratio. Furthermore, it compresses around twice as fast as MPPM, between 2.5 to 21 times faster than Re-pair, and around 2.3-4.5 times faster than p7zip. Yet, the heavy PPM modeling cost makes it around 10-15 times slower than Re-pair and 9-15 times slower than p7zip at decompression.

The case of Re-pair is the only exception of our experiments. The preprocessing can only improve the compression speed, yet it is still very slow.

7.3. Boosting Online Search

Dense codes were designed as text compressors that allow direct search and random access. They are not the best techniques in compression ratio, although they can compete with many state-of-the-art compressors. Their success is their good trade-off between compression ratio and speed during compression and decompression, and more importantly, their ability to directly search the compressed text faster than searching the plain text.

Although in the next section we will show that self-indexes take advantage from indexing a TH, SCBDC, and SCDC compressed text, this section is devoted to online search. We show that if a backend compressor produces compressed text that allows reasonably efficient decompression, then the preprocessing step also boosts the search on the compressed file up to the point that, in some cases, it is even faster than searching the original file, yet with much better compression ratios than ETDC or SCDC alone.
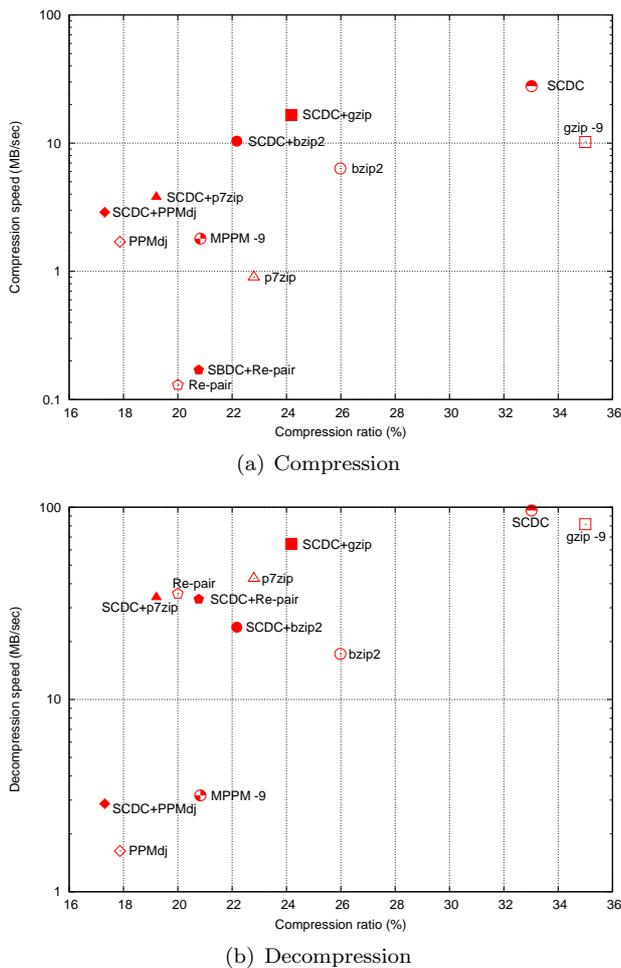
(a) Compression

(b) Decompression

FIGURE 5. Space vs compression/decompression time trade-offs on corpus ALL. The x axis shows compression ratio (leftwards is better). The y axis (in logarithmic scale) shows compression and decompression speed (MB/sec), respectively (upper is better).

The reason for this achievement is that the backend decompressor must generate only about one third of the text than if applied without preprocessing. Over this resulting decompressed text, we run the ETDC or SCDC searches, which are several times faster than on the original uncompressed text files. Two factors benefit the search over the preprocessed text: *i)* a shorter input file; and *ii)* a larger alphabet (with a probability of $1/119.4 \approx 0.008$ that two random characters match, versus $1/19.3 \approx 0.052$ on plain text), which produces fewer unsuccessful partial matchings during the search.

From the list of backend compressors of this empirical study, we chose gzip, which is the fastest one at decompression. We note that, coupled with ETDC or SCDC preprocessing, it achieves a very decent compression ratio of around 24% on large collections. We used software *Lzgrep*[27] [54], which couples the

---

[27]http://www.dcc.uchile.cl/~gnavarro/software/lzgrep.tar.gz.

decompression and the pattern matching on a single buffer, and modified the original software in order to search text compressed with ETDC and SCDC instead of plain text.[28]

We performed single and multi-pattern searches for patterns chosen at random over corpus ALL. We compared searches over dense+gzip with searches over uncompressed text, over text compressed with ETDC, and over text compressed with gzip.

To search text compressed with ETDC, we use our own implementations of *Horspool* and *Set-Horspool* algorithms[29] [55, 56]: *Horspool* for single-pattern searches and *Set-Horspool* for multi-pattern searches.

Three different algorithms were tested to search the uncompressed text: our own implementations of *Horspool* and *Set-Horspool* algorithms, and the *Agrep*[30] software [57]. *Agrep* returns chunks of text containing one or more searched patterns. The default chunk is a line. When traversing a chunk, if *Agrep* finds a search pattern, it skips the processing of the rest of the chunk. This appreciably distorts the comparison with the rest of the searchers. To present a fairer comparison, we performed the searches over a modified version of the text obtained by removing all the pattern occurrences from it, and then scaled the results. More precisely, we computed the text T' that is obtained by removing all the pattern occurrences from the original text. Then we ran *Agrep -s* over T' and scaled the resulting times assuming that $|T'| = |ALL|$. This shows essentially the same statistics and reflects more accurately the real search cost of *Agrep*.

Finally, we used the original Lzgrep to search text compressed with gzip. All the software of this experiment was compiled with the -m32 flag.

To choose the search patterns, we considered the vocabulary of corpus ALL, and extracted sets of patterns with K words of length L at random. We considered lengths *L=5* and *10,* and sets of *K=1, 5, 10, 25, 50, 100, 500*, and *1000* patterns. Figure 6 shows the average running times of the searchers over 10 different sets for each combination of L and K.

Leaving apart ETDC, which is unbeatable in search speed, by comparing Lzgrep over ETDC+gzip or SCDC+gzip with the plain text searchers, we observe that the search over that compressed text is between 1.5 and 10 times slower in the case of patterns of length 10. When searching for 100 patterns or less of length 5 (the average length of English words), where the plain text searchers perform shorter shifts, Lzgrep over dense+gzip is still 6-468% slower. However, when more than 100 patterns are searched for in parallel, the cascade is able to overcome plain text searchers by around 6-33%.
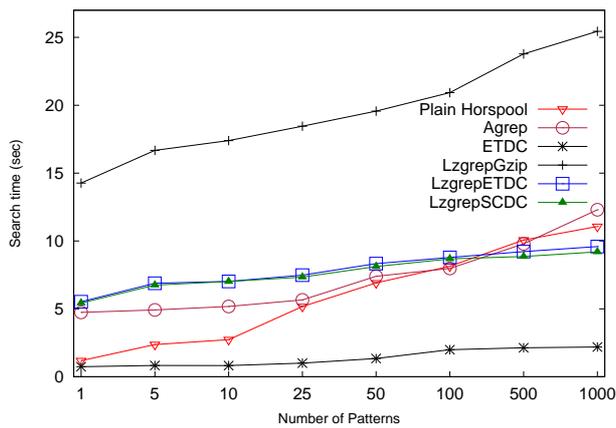
The preprocessing is indeed successful, as Lzgrep

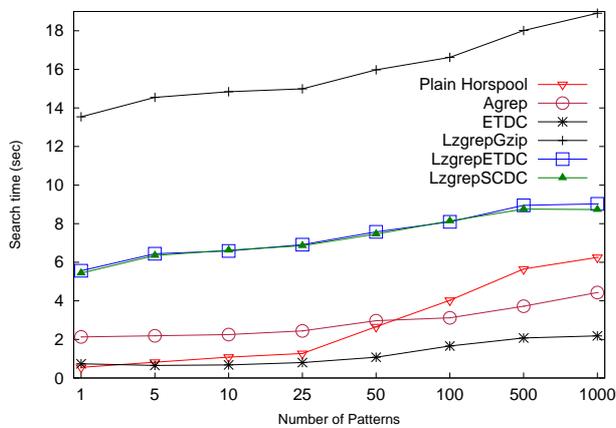[28]These modified versions of Lzgrep are available at http://vios.dc.fi.udc.es/codes

[29]http://vios.dc.fi.udc.es/codes.

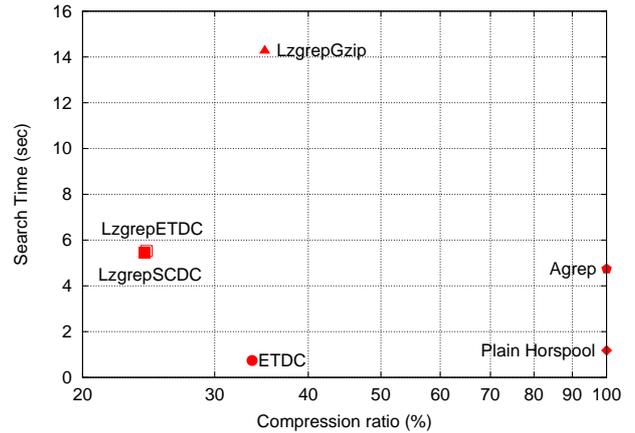[30]ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z.

(a) Patterns length 5.

(b) Patterns length 10.

FIGURE 6. Search times with patterns of different lengths. The x axis shows the number of patterns and the y axis plots the search time (lower is better).
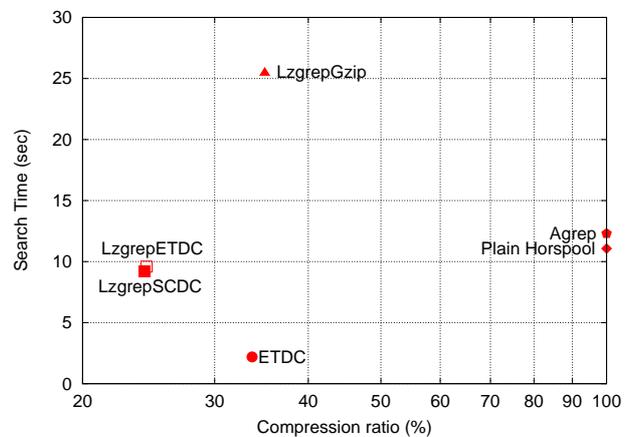

(a) 1 Pattern length 5.

(b) 1000 Patterns length 5.

FIGURE 7. Space vs online search speed trade-offs on corpus ALL. The x axis shows compression ratio in logarithmic scale (leftwards is better) and the y axis plots search time (lower is better).

over dense+gzip is between 2 and 2.5 times faster than Lzgrep over text compressed with gzip. Note also that the original Lzgrep was not competitive with uncompressed text searchers, being preferable to decompress and then search. However, with the cascades, the gap between Lzgrep and the plain text searchers narrows and, as seen, Lzgrep is able to overcome the others in some cases.

7.3.1. Discussion

Figure 7 shows the trade-offs between space and search speed achieved by the Lzgrep searchers over text compressed with dense+gzip in comparison with those searchers working over either plain text or text compressed with ETDC. We used patterns of length 5, the average length of English words. As seen, Lzgrep over dense compressed text obtains an interesting trade-off, with the best compression ratios of the comparative (around 24%), Lzgrep over dense+gzip obtains results comparable to those of the plain text searchers and, as explained, an important improvement with respect to

the original Lzgrep. With compression ratios around 33%, the searches over ETDC compressed text remain as the fastest ones, being around 4-11 times faster than Lzgrep over dense+gzip.

7.4. Boosting Indexed Search

Figures 8, 9, and 10 compare the AFFM and CSA self-indexes[31] built over text compressed with TH, SCBDC, and SCDC with those built over the original text. Throughout this section SCDC refers to the version presented in Section 6.3, that is, the one that outputs the compressed text backwards. We also include both WTBC and WCSA in the comparison. Although we run the same experiments on SSA and LZI, we omit them to avoid cluttering the figures, as they were not competitive with AFFM and CSA.

Using corpus CR, and different values of the tuning parameters of each self-index, the figures show the

---

[31]Software available at `http://vios.dc.fi.udc.es/indexing`.

space/time trade-off obtained for *locating* and *counting* all the occurrences of patterns composed of 1 and 4 words. We also show *extract* performance for each self-index. We measure the time needed to recover 1,000 snippets of the original text with 10,000 characters each. We give average times per extracted character. We do not include *display* operation, as it is basically locate plus extract.

In all cases, the cascades clearly improve the space/time trade-off obtained by their traditional counterparts. Furthermore, we compared the size of the text compressed by the backend compressors with that of both AFFM and CSA tuned to obtain low space requirements. In the case of CSA when reducing the sample rate (SR) to 256 and setting the sampling of Ψ (S_Ψ) to 256, the resulting SCBDC+CSA achieves a compression ratio of 31.64%, that is, even better than applying ETDC alone (31.94%), just as the k-th order compressors improved upon the result of ETDC(T) or SCDC(T). In addition, SCDC+CSA with SR=1024 and S_Ψ=1024 obtains a compression ratio of 30.33%, hence overcoming the compression ratio of SCDC (31.29%) by around 1 percentage point. Similarly, the 33.52% achieved by the AFFM built over text compressed with SCBDC when using SR=4096 and the rank factor (RF) is set to 64, still overcomes the compression ratio of TH alone (34.28%) and also that of SCBDC (33.93%). However, the AFFM built on top of SCDC was not able to overcome compressors alone. Even though we tuned it with a very sparse sampling (SR=16384 and RF=1024), it obtained a compression ratio of 35.96%.

In any case, if we set the parameters to obtain structures of the same size, the self-indexes built over compressed text will always be much faster. For example, the AFFM over plain text with SR=1024 and RF=64 achieves a compression ratio of 56.94%. Using AFFM over SCBDC, to obtain a structure of similar size, we can set SR=16 and RF=4 (55.77%). In these circumstances, the self-index built over text compressed with SCBDC is 80-130 times faster to perform *locate*, around 78%-100% faster at performing *count*, and around 2.5 times faster on *extract* operation. On the other hand, if we set the parameters to obtain two structures that achieve similar search times, the self-index over compressed text will occupy between half and one third of the size of the index built from plain text. For example, AFFM over plain text with SR=16 and RF=64 obtained a locating time per occurrence of 3.609 μsec, when searching for patterns of 4 words, whereas the AFFM over TH with SR=16 and RF=8, obtains 3.601 μsec. Yet, the self-index over TH occupies 55.67% of the original text, whereas the self-index over plain text occupies 106.16%. Using again the AFFM over plain text with SR=16 and RF=64, its counting time for patterns of length 4 is 0.0224 msec; to achieve a similar time with AFFM over TH, we have to set RF=64 and SR=1024 (0.0214 msec). With these
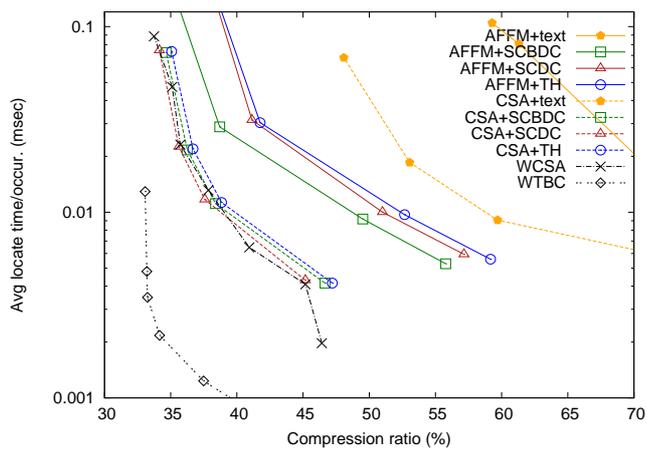
settings, the AFFM over TH occupies 36.69%, whereas AFFM over plain text occupies 106.16%. Finally, AFFM over plain text with SR=16 and RF=4 needs 0.565 μsec to extract one character (on average), and its compression ratio is 116.36%. AFFM over TH with RF=32 and SR=1024 consumes 0.546 μsec, with a compression ratio of 37.10%. Similar conclusions are obtained when comparing the indexes built on top of SCDC with those built over plain text.

With respect to the differences between TH and SCBDC, the k-th order self-index AFFM seems to predict better the bytes in SCBDC(T) than those in TH(T). This is not surprising: after a one-byte value or a stopper only a one-byte value or a beginner can follow, after a beginner only a continuer or a stopper can follow, and so on; this is not that regular on TH. With the same parameters, the index over SCBDC is around 6-9% smaller than the one built over text compressed with TH. The same happens with CSA, but differences are only around 1%.
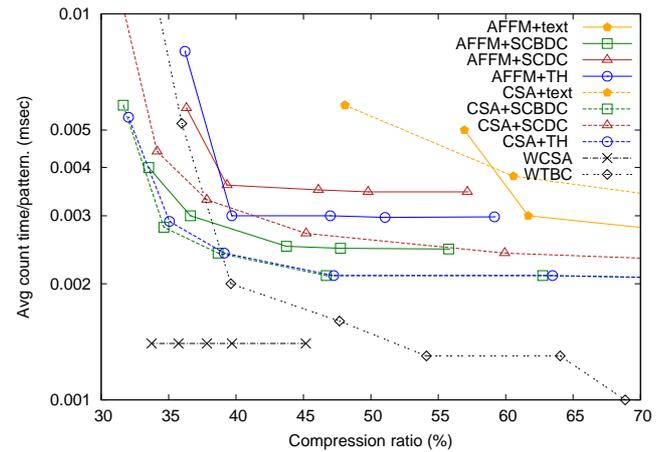
When we compare SCBDC with SCDC, we find that, surprisingly, the AFFM built on top of SCBDC obtains much better compression than that over SCDC (around 3 percentage points in most cases when setting both indexes with the same parameters). Furthermore, the SCBDC version clearly obtains also better performance at both *count* and *locate* operations, whereas the times for *extract* are similar when using both SCDC and SCBDC. In the case of CSA, results show that CSA built on top of SCDC requires slightly less memory to work. The SCDC version is slower at *count* than that using SCBDC, but it becomes faster at *extract*. In *locate*, CSA built on top of both SCDC and that built over SCBDC obtain similar times.

With respect to WTBC and WCSA, the trade-off between compression ratio and *locate* performance of our CSA self-index built over compressed text is comparable to that of WCSA, although worse than that of WTBC in short patterns. When considering the *count* performance for short patterns, CSA over compressed text can compete with WTBC, but its results are worse than those of the WCSA. Yet, in long patterns, CSA over compressed text is able to overcome the WCSA and improves by far the results of the WTBC. Finally, in the *extract* operation, the character-based self-indexes over compressed text are clearly worse than both WCSA and WTBC. This is expectable as the last ones extract data word-wise, so each extracted symbol consists in one word rather than in just one single character.
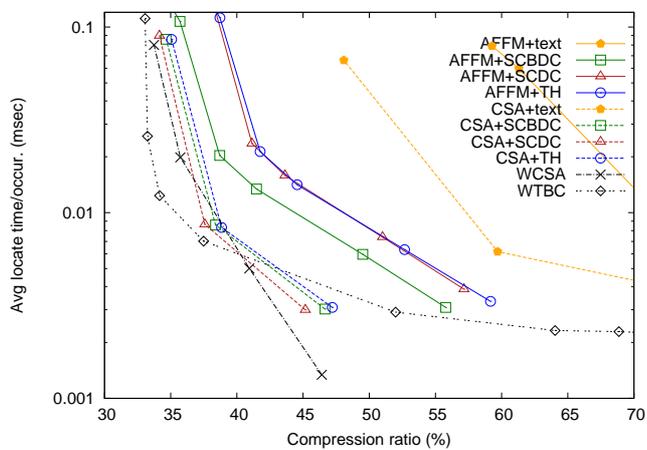
We remind that WCSA and WTBC are very sophisticated indexing systems, so it is remarkable that our simple combinations perform reasonably close by just prepending a word encoding to the character-based indexers.
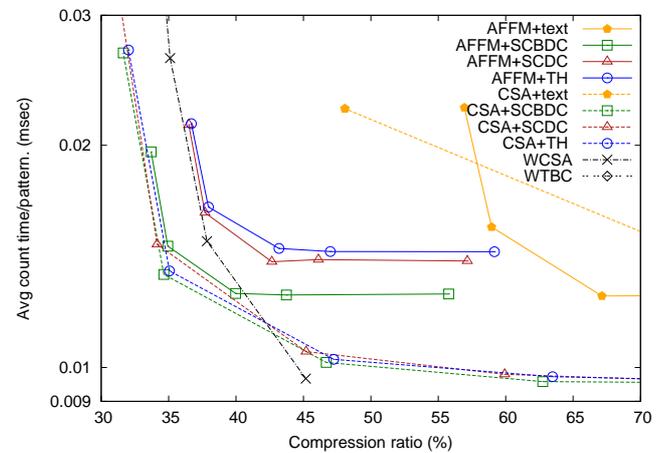
(a) Locate Patterns 1 word

(a) Count Patterns 1 word

(b) Locate Patterns 4 words

(b) Count Patterns 4 words

FIGURE 8. Locate space/search time trade-offs on corpus CR. The x axis shows compression ratio (leftwards is better), and y axis plots average search time per occurrence in logarithmic scale (lower is better).

FIGURE 9. Count space/search time trade-offs on corpus CR. The x axis shows compression ratio (leftwards is better), and y axis plots average count time per occurrence in logarithmic scale (lower is better). The plot for WTBC is not shown in the bottom figure because its times are always over 0.5 msec.

8.   CONCLUSIONS

We have shown that byte-oriented natural language compressors such as ETDC, SCDC, SCBDC, 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, in the case of TH and SCBDC (and in some cases SCDC), can also be self-indexed. The results obtained by cascading word-based byte-oriented semistatic compressors with the block-wise bzip2, the Ziv-Lempel-based gzip and p7zip (LZMA), and the predictive PPM-based PPMdj, show clear improvements in their compression ratio and speed. In the case of bzip2 and PPMdj (the slowest techniques

at decompression) the decompression speed was also improved.

We showed that the cascades yield an attractive space/efficiency trade-off. Dense+gzip is very fast, obtains very good compression, and allows quite efficient direct search over the compressed text. Dense+bzip2 compresses a bit more but it is also slower. Dense+p7zip obtains even better compression at the expense of losing some performance. Dense+PPMdj obtains the best compression ratio of our experiments, around an impressive 17%. Finally, Re-pair remained as the only compressor that does not take a clear advantage of the preprocessing.

We also showed that the preprocessing improves considerably the search times on the compressed text. We empirically demonstrated this over text compressed with gzip. We adapted the Lzgrep software to
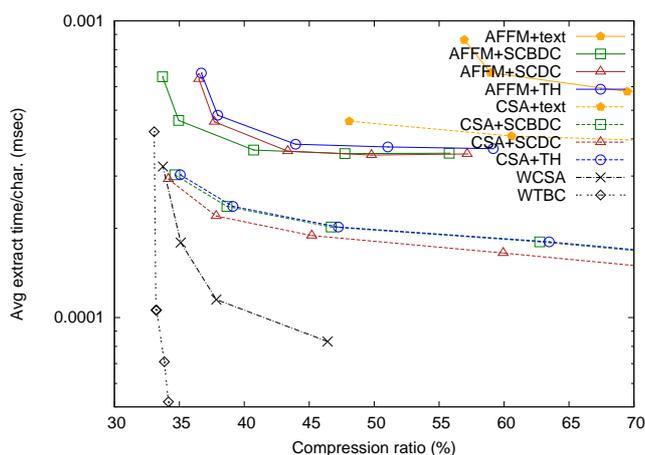
FIGURE 10. Extract space/search time trade-offs on corpus CR. The x axis shows compression ratio (leftwards is better), and y axis plots average extract time per char in logarithmic scale (lower is better).

search both text compressed with ETDC+gzip and SCDC+gzip. The empirical results showed that the searches over text compressed with dense+gzip are around 2-2.5 times faster than those performed with the original Lzgrep over text compressed with gzip. Moreover, when we search for more than 100 patterns of length 5, Lzgrep over dense+gzip outperforms the plain text searchers.

We have seen that self-indexing can be boosted in a similar way by just building the traditional char-based self-indexes over the compressed sequence obtained with a prefix and suffix-free word-based byte-oriented text compressor. As the only candidate up to now was TH, we also developed SCBDC, the first suffix-free compressor from the family of dense codes. SCBDC retains the same features of TH. Yet, it is simpler to implement and improves its compression ratio, while obtaining similar performance. We also showed how SCDC codes can be used in combination with some self-indexes to provide indexed searches, despite not being suffix-free.

Our experiments showed how indexing also benefits from the use of a compressed version of the text. If we build both the AFFM and CSA over text compressed with TH, SCBDC or SCDC, and we set their parameters in order to obtain structures of the same size as if we indexed the plain text, we obtain self-indexes that are much (up to 130 times) faster than the traditional ones. If we instead set the parameters of each self-index to obtain two structures with similar search speed, the self-index built over compressed text will occupy between half and one third of the size of that built over plain text. The newer WTBC and WCSA improve our results, but these self-indexes represent new complex developments, whereas our results are

obtained combining already developed software and adding a small amount of new code.

As future work, we are interested in obtaining actual implementations of joined ETDC+X or SCDC+X compressors and decompressors, especially in the case of ETDC+gzip because of its impressive compression/performance trade-off. These joined versions would avoid many disk I/Os, thus further improving the performance.

In addition, in the case where efficient direct search is not possible due to the backend compressor, we can exploit other byte-oriented codes that display better compression performance [21]. Under the same circumstances, we can explore the use of vocabulary pruning techniques [43, 45] in order to avoid the appearance of 3-byte codewords, and then obtain better compression ratios.

FUNDING

REFERENCES

[1] Fariña, A., Navarro, G., and Paramá, J. (2008) Word-based statistical compressors as natural language compression boosters. *Proceedings of Data Compression Conference (DCC 08)*, Snowbird, UT, 25-27 March, pp. 162–171. IEEE Computer Society, Los Alamitos, CA.

[2] Huffman, D. A. (1952) A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 8 September, pp. 1098–1101. Institute of Radio Engineers Inc., New York, NY.

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

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

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

[6] Abramson, N. (1963) *Information Theory and Coding.* McGraw-Hill, New York, NY.

[7] Jelinek, F. (1968) *Probabilistic Information Theory.* McGraw-Hill, New York, NY.

[8] Witten, I., Neal, R., and Cleary, J. (1987) Arithmetic coding for data compression. *Communications of the ACM*, **30**, 520–541.

[9] Burrows, M. and Wheeler, D. J. (1994) A block-sorting lossless data compression algorithm. Technical Report 124. Digital Equipment Corporation, Palo Alto, CA.

[10] Larsson, N. J. and Moffat, A. (1999) Offline dictionary-based compression. *Proceedings of Data Compression*

Conference (DCC 99), Snowbird, UT, 29-31 March, pp. 296–305. IEEE Computer Society, Los Alamitos, CA.

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

[12] Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. (1986) A locally adaptive data compression scheme. *Communications of the ACM*, **29**, 320–330.

[13] Zipf, G. K. (1949) *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Cambridge, MA.

[14] Heaps, H. (1978) *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, New York, NY.

[15] Moura, E., Navarro, G., Ziviani, N., and Baeza-Yates, R. (2000) Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, **18**, 113–139.

[16] Klein, S. and Shapira, D. (2005) Pattern matching in Huffman encoded texts. *Information Processing and Management*, **41**, 829–841.

[17] Ferguson, T. and Rabinowitz, J. (1984) Self-synchronizing Huffman codes. *IEEE Transactions on Information Theory*, **30**, 687–697.

[18] Biskup, M. (2008) Guaranteed synchronization of Huffman codes. *Proceedings of Data Compression Conference (DCC 08)*, Snowbird, UT, 25-27 March, pp. 462–471. IEEE Computer Society, Los Alamitos, CA.

[19] Biskup, M. and Plandowski, W. (2009) Guaranteed synchronization of Huffman codes with known position of decoder. *Proceedings of Data Compression Conference (DCC 09)*, Snowbird, UT, 16-18 March, pp. 33 –42. IEEE Computer Society, Los Alamitos, CA.

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

[21] Culpepper, J. and Moffat, A. (2005) Enhanced byte codes with restricted prefix properties. *Proceedings of the 12th Conference on String Processing and Information Retrieval (SPIRE 05)*, Buenos Aires, Argentina, 2-4 November LNCS 3772, pp. 1–12. Springer, Berlin / Heidelberg.

[22] Klein, S. T. and Ben-Nissan, M. K. (2008) Using Fibonacci compression codes as alternatives to dense codes. *Proceedings of Data Compression Conference (DCC 08)*, Snowbird, UT, 25-27 March, pp. 472–481. IEEE Computer Society, Los Alamitos, CA.

[23] Klein, S. and Shapira, D. (2011) The string-to-dictionary matching problem. *Proceedings of Data Compression Conference (DCC 11)*, Snowbird, UT, 29-31 March, pp. 143–152. IEEE Computer Society, Los Alamitos, CA.

[24] Navarro, G. and Mäkinen, V. (2007) Compressed full-text indexes. *ACM Computing Surveys*, **39**, article 2, 61 pages.

[25] Sadakane, K. (2003) New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, **48**, 294–313.

[26] Mäkinen, V. and Navarro, G. (2005) Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, **12**, 40–66.

[27] Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. (2007) Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, **3**, article 20, 24 pages.

[28] Navarro, G. (2004) Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, **2**, 87–114.

[29] Arroyuelo, D. and Navarro, G. (2010) Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. *ACM Journal of Experimental Algorithmics*, **15**, article 1.5, 54 pages.

[30] Brisaboa, N., Fariña, A., Navarro, G., Places, A., and Rodríguez, E. (2008) Self-indexing natural language. *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE 08)*, Melbourne, Australia, 10-12 November LNCS 5280, pp. 121–132. Springer, Berlin / Heidelberg.

[31] Brisaboa, N., Fariña, A., Ladra, S., and Navarro, G. (2008) Reorganizing compressed text. *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 08)*, Singapore, 20-24 July, pp. 139–146. ACM, New York, NY.

[32] Moffat and Turpin (1997) On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, **45**, 1200–1207.

[33] Horspool, R. N. and Cormack, G. V. (1992) Constructing word-based text compression algorithms. *Proceedings of Data Compression Conference (DCC 1992)*, Snowbird, UT, 24-27 March, pp. 62–71. IEEE Computer Society Press, Los Alamitos, CA.

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

[35] Isal, R. and Moffat, A. (2001) Parsing strategies for BWT compression. *Proceedings of Data Compression Conference (DCC 01)*, Snowbird, UT, 27-29 March 2001, pp. 429–438. IEEE Computer Society, Los Alamitos, CA.

[36] Isal, R., Moffat, A., and Ngai, A. C. H. (2002) Enhanced word-based block-sorting text compression. *Twenty-Fifth Australasian Computer Science Conference*, Melbourne, Australia, January/February, CR-PIT, **4**, pp. 129–137. Australian Computer Society, Darlinghurst, Australia.

[37] Franceschini, R. and Mukherjee, A. (1996) Data compression using encrypted text. *Proceedings of Advanced Digital Libraries (ADL 96)*, Washinton, DC, 13-15 May, pp. 130–138. IEEE Computer Society, Los Alamitos, CA.

[38] Teahan, W. J. and Cleary, J. G. (1996) The entropy of english using PPM-based models. *Proceedings of Data Compression Conference (DCC 96)*, Snowbird, UT, 31 March- 3 April, pp. 53–62. IEEE Computer Society, Los Alamitos, CA.

[39] Awan, F., Zhang, N., Motgi, N., Iqbal, R., and Mukherjee, A. (2001) LIPT: A reversible lossless text transform to improve compression performance. *Proceedings of Data Compression Conference (DCC 01)*, Snowbird, UT, 27-29 March, pp. 481–481. IEEE Computer Society, Los Alamitos, CA.

[40] Sun, W., Zhang, N., and Mukherjee, A. (2003) Dictionary-based fast transform for text compression. *Proceedings of the International Conference on Information Technology: Computers and Communications*

(ITCC 03), Las Vegas, NV, 28-30 April, pp. 176–182. IEEE Computer Society Press, Los Alamitos, CA.

[41] Abel, J. and Teahan, W. (2005) Universal text preprocessing for data compression. *IEEE Transactions on Computers*, **54**, 497–507.

[42] Skibinski, P., Grabowski, S., and Deorowicz, S. (2005) Revisiting dictionary-based compression. *Software Practice and Experience*, **35**, 1455–1476.

[43] Adiego, J. and de la Fuente, P. (2006) Mapping words into codewords on PPM. *Proceedings of the 13th Conference on String Processing and Information Retrieval (SPIRE 06)*, Glasgow, UK, 11-13 October LNCS 4209, pp. 181–192. Springer, Berlin / Heidelberg.

[44] Shkarin, D. (2002) PPM: One step to practicality. *Proceedings of Data Compression Conference (DCC 02)*, Snowbird, UT, 2-4 April, pp. 202–211. IEEE Computer Society Press, Los Alamitos, CA.

[45] Adiego, J., Martinez-Prieto, M., and Fuente, P. (2009) High performance word-codeword mapping algorithm on PPM. *Proceedings of Data Compression Conference (DCC 09)*, Snowbird, UT, 16-18 March, pp. 23–32. IEEE Computer Society, Los Alamitos, CA.

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

[47] Ferragina, P. (2008) String algorithms and data structures. *CoRR*, **abs/0801.2378**.

[48] Bell, T. C., Cleary, J. G., and Witten, I. H. (1990) *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.

[49] Williams, H. E. and Zobel, J. (1999) Compressing integers for fast file access. *The Computer Journal*, **42**, 193–201.

[50] Fariña, A. (2005) New Compression Codes for Text Databases. PhD thesis Database Laboratory, University of A Coruña. Available at http://lbd.udc.es/Repository/Thesis/phdFarinha.pdf.

[51] Moura, E., Navarro, G., Ziviani, N., and Baeza-Yates, R. (1998) Fast searching on compressed text allowing errors. *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 98)*, Melbourne, Australia, 24-28 August, pp. 298–306. ACM, New York, NY.

[52] Wan, R. (2003) Browsing and Searching Compressed Documents. PhD thesis Dept. of Computer Science and Software Engineering, University of Melbourne. Avaliable at http://eprints.unimelb.edu.au/archive/00000871.

[53] Manber, U. and Myers, G. (1993) Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, **22**, 935–948.

[54] Navarro, G. and Tarhio, J. (2005) LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text. *Software Practice and Experience*, **35**, 1107–1130.

[55] Horspool, R. N. (1980) Practical fast searching in strings. *Software Practice and Experience*, **10**, 501–506.

[56] Navarro, G. and Raffinot, M. (2002) *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, New York, NY.

[57] Wu, S. and Manber, U. (1992) Agrep – a fast approximate pattern-matching tool. *Proceedings of the USENIX Winter 1992 Technical Conference (USENIX 92)*, San Francisco, CA, 20-24 January, pp. 153–162. USENIX, Berkeley, CA.

[58] Awan, F. and Mukherjee, A. (2001) LIPT: A lossless text transform to improve compression. *Proceedings of the International Conference on Information and Theory: Coding and Computing (ITCC 01)*, Las Vegas, NV, 2-4 April, pp. 452–460. IEEE Computer Society, Los Alamitos, CA.

[59] Chapin, B. and Tate, S. R. (1998) Higher compression from the Burrows-Wheeler transform by modified sorting. *Proceedings of Data Compression Conference (DCC 98)*, Snowbird, UT, 30 March–1 April, pp. 532–532. IEEE Computer Society, Los Alamitos, CA.

[60] Mäkinen, V. and Navarro, G. (2004) Compressed compact suffix arrays. *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM 04)*, Istambul, Turkey, 5-7 July LNCS 3109, pp. 420–433. Springer, Berlin / Heidelberg.