Lightweight Natural Language Text Compression *

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

[1] Database Lab., Univ. da Coruña, Facultade de Informática,
Campus de Elviña s/n, 15071 A Coruña, Spain.
{brisaboa,fari,parama}@udc.es

[2]Center for Web Research, Dept. of Computer Science, Univ. de Chile,
Blanco Encalada 2120, Santiago, Chile.
gnavarro@dcc.uchile.cl

Abstract

Variants of Huffman codes where words are taken as the source symbols are currently the most attractive choices to compress natural language text databases. In particular, Tagged Huffman Code by Moura et al. offers fast direct searching on the compressed text and random access capabilities, in exchange for producing around 11% larger compressed files. This work describes End-Tagged Dense Code and (s, c)-Dense Code, two new semistatic statistical methods for compressing natural language texts. These techniques permit simpler and faster encoding and obtain better compression ratios than Tagged Huffman Code, while maintaining its fast direct search and random access capabilities. We show that Dense Codes improve Tagged Huffman Code compression ratio by about 10%, reaching only 0.6% overhead over the optimal Huffman compression ratio. Being simpler, Dense Codes are generated 45% to 60% faster than Huffman codes. This makes Dense Codes a very attractive alternative to Huffman code variants for various reasons: they are simpler to program, faster to build, of almost optimal size, and as fast and easy to search as the best Huffman variants, which are not so close to the optimal size.

Keywords: Text databases, natural language text compression, searching compressed text.

1 Introduction

Text compression (Bell et al. 1990) permits representing a document using less space. This is useful not only to save disk space, but more importantly, to save disk transfer and network transmission time. In recent years, compression techniques especially designed for natural language texts have not only proven extremely effective (with compression ratios[1] around 25%-35%), but also permitted searching the compressed text much faster (up to 8 times) than the original text (Turpin and Moffat 1997, Moura et al. 1998, Moura et al. 2000). The integration of compression and indexing techniques (Witten et al. 1999, Navarro et al. 2000, Ziviani et al. 2000) opened the door to *compressed text databases*, where texts and indexes are manipulated directly in compressed form, thus saving both time and space.

Not every compression method is suitable for a compressed text database. The compressed text should satisfy three basic conditions: (1) it can be directly accessed at random positions, (2) it can be decompressed fast, and (3) it can be searched fast. The rationale of conditions (1) and (2) is pretty obvious, as one wishes to

*Supported by CYTED VII.19 RIBIDI Project and (for the third author) Millennium Nucles Center for Web Research, Grant P04-67-F, Mideplan, Chile. Also funded (for the Spanish group) by MCyT (PGE and FEDER) grant(TIC2003-06593) and Xunta de Galicia grant(PGIDIT05SIN10502PR).

[1]The size of the compressed file as a percentage of the original size.

display individual documents to final users without need of decompressing the whole text collection preceding it. Moreover, in many cases it is necessary to display only a snippet of the text around an occurrence position, and thus it must be possible to start decompression from any position of the compressed text, not only from the beginning of a document or even from the beginning of a codeword. Condition (3) could be unnecessary if an inverted index pointing to all text words were available, yet such indexes take up significant extra space (Baeza-Yates and Ribeiro-Neto 1999). Alternatively, one can use inverted indexes pointing to whole documents, which are still able of solving one-word queries without accessing the text. Yet, more complex queries such as phrase and proximity queries will require some direct text processing in order to filter out documents containing all the query terms in the wrong order. Moreover, some space-time tradeoffs in inverted indexes are based on grouping documents into blocks, and therefore sequential scanning is necessary even on single-word queries (Manber and Wu 1994, Navarro et al. 2000). Although partial decompression followed by searching is a solution, direct search of the compressed text is much more efficient (Ziviani et al. 2000).

Classic compression techniques are generally unattractive for compressing text databases. For example, the well-known algorithms of Ziv and Lempel (1977, 1978) permit searching the text directly, without decompressing it, in half the time necessary for decompression (Navarro and Tarhio 2000, Navarro and Tarhio 2005). Yet, as any other adaptive compression technique, it does not permit direct random access to the compressed text, thus failing on Condition (1). Semistatic techniques are necessary to ensure that the decoder can start working from any point of the compressed text without having seen all the previous text. Semistatic techniques also permit fast direct search of the compressed text, by (essentially) compressing the pattern and searching for it in the compressed text. This does not work on adaptive methods, as the pattern does not appear in the same form across the compressed text.

Classic semistatic compression methods, however, are not entirely satisfactory either. For example, the Huffman (1952) code offers direct random access from codeword beginnings and decent decompression and direct search speeds (Miyazaki et al. 1998), yet the compression ratio of the Huffman code on natural language is poor (around 65%).

The key to the success of natural language compressed text databases is the use of a semistatic *word-based* model by Moffat (1989), so that the text is regarded as a sequence of words (and separators). A word-based Huffman code (Turpin and Moffat 1997), where codewords are sequences of bits, achieves 25% of compression ratio, although decompression and search are not so fast because of the need of bit-wise manipulations. A byte-oriented word-based Huffman code, called *Plain Huffman Code (PHC)* by Moura et al. (2000), eliminates this problem by using 256-ary Huffman trees, so that codewords are sequences of bytes. As a result, decompression and search are faster, although compression ratios rise to 30%. As the compression ratio is still attractive, they also propose *Tagged Huffman Code (THC)*, whose compression ratio is around 35% but permits much faster Boyer-Moore-type search directly in the compressed text, as well as decompression from any point of the compressed file (even if not codeword-aligned).

In this paper, we improve the existing tradeoffs on word-based semistatic compression, presenting two new compression techniques that allow direct searching and direct access to the compressed text. Roughly, we achieve the same search performance and capabilities of Tagged Huffman Code, combined with compression ratios similar to those of Plain Huffman Code. Our techniques have the additional attractiveness of being very simple to program.

We first introduce *End-Tagged Dense Code* (ETDC), a compression technique that allows (i) efficient decompression of arbitrary portions of the text (direct access), and (ii) efficient Boyer-Moore-type search directly on the compressed text, just as in Tagged Huffman Code. End-Tagged Dense Code improves both Huffman codes in encoding/decoding speed. It also improves Tagged Huffman Code in compression ratio, while retaining similar search time and capabilities.

We then present *(s,c)-Dense Code* (SCDC), a generalization of End-Tagged Dense Code which achieves better compression ratios while retaining all the search capabilities of End-Tagged Dense Code. (s,c)-Dense Code poses only a negligible overhead over the optimal[2] compression ratio reached by Plain Huffman Code.

[2]Optimal among all semistatic prefix-free byte-oriented codes.

Partial early versions of this paper were presented in (Brisaboa et al. 2003a, Brisaboa et al. 2003b, Fariña 2005).

The outline of this paper is as follows. Section 2 starts with some related work. Section 3 presents our first technique, End-Tagged Dense Code. Next, Section 4 introduces the second technique, (s, c)-Dense Code. Encoding, decoding, and search algorithms for both compression techniques are presented in Section 5. Section 6 is devoted to empirical results. Finally, Section 7 gives our conclusions and future work directions.

2 Related work

Text compression (Bell et al. 1990) consists of representing a sequence of characters using fewer bits than its original representation. The text is seen as a sequence of *source symbols* (characters, words, etc.). For the reasons we have explained, we are interested in *semistatic* methods, where each source symbol is assigned a *codeword* (that is, a sequence of *target symbols*), and this assignment does not change across the compression process. The compressed text is then the sequence of codewords assigned to its source symbols. The function that assigns a codeword to each source symbol is called a *code*. Among the possible codes, *prefix codes* are preferable in most cases. A *prefix code* guarantees that no codeword is a prefix of another, thus permitting decoding a codeword right after it is read (hence the alternative name *instantaneous code*).

The Huffman (1952) code is the optimal (shortest total length) prefix code for any frequency distribution. It has been traditionally applied to text compression by considering characters as source symbols and bits as the target symbols. On natural language texts, this yields poor compression ratios (around 65%). The key idea to the success of semistatic compression on natural language text databases was to consider words as the source symbols (Moffat 1989) (as well as separators, defined as maximal text substrings among consecutive words). The distribution of words in natural language is much more skewed than that of characters, following a Zipf Law (that is, the frequency of the i-th most frequent word is proportional to $1/i^\theta$, for some $1 < \theta < 2$ (Zipf 1949, Baeza-Yates and Ribeiro-Neto 1999)), and the separators are even more skewed. As a result, compression ratios get around 25%, which is close to what can be obtained with any other compression method (Bell et al. 1990). The price of having a larger set of source symbols (which semistatic methods must encode together with the compressed text) is not significant on large text collections, as the vocabulary grows slowly ($O(N^\beta)$ symbols on a text of N words, for some $\beta \approx 0.5$, by Heaps Law (Heaps 1978, Baeza-Yates and Ribeiro-Neto 1999)).

This solution is acceptable for compressed text databases. With respect to searching those Huffman codes, essentially one can compress the pattern and search the text for it (Turpin and Moffat 1997, Miyazaki et al. 1998). However, it is necessary to process the text bits sequentially in order to avoid *false matches*. Those occur because the compressed pattern might appear in a text not aligned to any codeword, that is, the concatenation of two codewords might contain the pattern bit string, yet the pattern is not in the text. A sequential processing ensures that the search is aware of the codeword beginnings and thus false matches are avoided.

With such a large source vocabulary, it makes sense to have a larger target alphabet. The use of bytes as target symbols was explored by Moura et al. (2000), who proposed two byte-oriented word-based Huffman codes as a way to speed up the processing of the compressed text.

The first, *Plain Huffman Code (PHC)*, is no more than a Huffman code where the source symbols are the text words and separators, and the target symbols are bytes. This obtains compression ratios close to 30% on natural language, a 5% of overhead with respect the word-based approach of Moffat (1989), where the target symbols are bits. In exchange, decompression and in general traversal of the compressed text is around 30% faster with Plain Huffman Code, as no bit manipulations are necessary (Moura et al. 2000). This is highly valuable in a compressed text database scenario.

The second code, *Tagged Huffman Code (THC)*, is similar except that it uses the highest bit of each byte to signal the first byte of each codeword. Hence, only 7 bits of each byte are used for the Huffman code. Note that the use of a Huffman code over the remaining 7 bits is mandatory, as the flag is not useful by

itself to make the code a prefix code. Compared to Plain Huffman Code, Tagged Huffman Code produces a compressed text around 11% longer reaching 35% of compression ratio.

There are two important advantages to justify this choice in a compressed text database scenario. First, Tagged Huffman Code can be accessed at any position for decompression, even in the middle of a codeword. The flag bit permits easy synchronization to the next or previous codeword. Plain Huffman Code, on the other hand, can start decompression only from codeword beginnings. Second, a text compressed with Tagged Huffman Code can be searched efficiently, by just compressing the pattern word or phrase and then running any classical string matching algorithm for the compressed pattern on the compressed text. In particular, one can use those algorithms able of skipping characters (Boyer and Moore 1977, Navarro and Raffinot 2002). This is not possible with Plain Huffman Code, because of the false matches problem. On Tagged Huffman Code false matches are impossible thanks to the flag bits.

It is interesting to point out some approaches that attempt to deal with the false matches problem without scanning every target symbol. The idea is to find a synchronization point, that is, a position in the compressed text where it is sure that a codeword starts. Recently, Klein and Shapira (2005) proposed that once a match of the search pattern is found at position i, a decoding algorithm would start at position $i-K$, being K a constant. It is likely that the algorithm synchronizes itself with the beginning of a codeword before it reaches again position i. However, false matches may still appear, and the paper analyzes the probability of reporting them as true matches.

Another alternative, proposed by Moura et al. (2000), is to align the codeword beginnings to block boundaries of B bytes. That is, no codeword is permitted to cross a B-byte boundary and thus one can start decompression at any point by going back to the last position multiple of B. This way, one can search using any string matching algorithm, and then has to rule out false matches by retraversing the blocks where matches have been found, in order to ensure that those are codeword-aligned. They report best results with $B = 256$, where they pay a space overhead of 0.78% over Plain Huffman Code and a search time overhead of 7% over Tagged Huffman Code.

Moura et al. (2000) finally show how more complex searches can be carried out. For example, complex patterns that match a single word are first searched for in the vocabulary, and then a multipattern search for all the codewords of the matching vocabulary words is carried out on the text. Sequences of complex patterns can match phrases following the same idea. It is also possible to perform more complex searches, such as approximate matching at the word level (that is, search for a phrase permitting at most k insertions, deletions, replacements, or transpositions of words). Overall, the compressed text not only takes less space than the original text, but it is also searched 2 to 8 times *faster*.

The combination of this compressed text with compressed indexes (Witten et al. 1999, Navarro et al. 2000) opens the door to compressed text databases where the text is always in compressed form, being decompressed only for presentation purposes (Ziviani et al. 2000).

Huffman coding is a *statistical* method, in the sense that the codeword assignment is done according to the frequencies of source symbols. There are also some so-called *substitution* methods suitable for compressed text databases. The earliest usage of a substitution method for direct searching we know of was proposed by Manber (1997), yet its compression ratios were poor (around 70%). This encoding was a simplified variant of Byte-Pair Encoding (BPE) (Gage 1994). BPE is a multi-pass method based on finding frequent pairs of consecutive source symbols and replacing them by a fresh source symbol. On natural language text, it obtains a poor compression ratio (around 50%), but its word-based version is much better, achieving compression ratios around 25%-30% (Wan 2003). It has been shown how to search the character-based version of BPE with competitive performance (Shibata et al. 2000, Takeda et al. 2001), and it is likely that the word-based version can be searched as well. Yet, the major emphasis in the word-based version has been the possibility of browsing over the frequent phrases of the text collection (Wan 2003).

Other methods with competitive compression ratios on natural language text, yet unable of searching the compressed text faster than the uncompressed text, include Ziv-Lempel compression (Ziv and Lempel 1977, Ziv and Lempel 1978) (implemented for example in *Gnu gzip*), Burrows-Wheeler compression

Rank	ETDC	THC
1	**1**00	**1**00
2	**1**01	**1**01
3	**1**10	**1**10
4	**1**11	**1**11 **0**00
5	**0**00 **1**00	**1**11 **0**01
6	**0**00 **1**01	**1**11 **0**10
7	**0**00 **1**10	**1**11 **0**11 **0**00
8	**0**00 **1**11	**1**11 **0**11 **0**01
9	**0**01 **1**00	**1**11 **0**11 **0**10
10	**0**01 **1**01	**1**11 **0**11 **0**11

Table 1: Comparative example among ETDC and THC, for b=3.

(Burrows and Wheeler 1994) (implemented for example in Seward's *bzip2*), and statistical modeling with arithmetic coding (Carpinelli et al. 1999).

3 End-Tagged Dense Code

We obtain *End-Tagged Dense Code (ETDC)* by a simple change to Tagged Huffman Code (Moura et al. 2000). Instead of using the highest bit to signal the *beginning* of a codeword, it is used to signal the *end* of a codeword. That is, the highest bit of codeword bytes is 1 for the *last* byte (not the first) and 0 for the others.

This change has surprising consequences. Now the flag bit is enough to ensure that the code is a prefix code regardless of the content of the other 7 bits of each byte. To see this, consider two codewords X and Y, where X is shorter than Y ($|X| < |Y|$). X cannot be a prefix of Y because the last byte of X has its flag bit set to 1, whereas the $|X|$-th byte of Y has its flag bit set to 0. Thanks to this change, there is no need at all to use Huffman coding in order to ensure a prefix code. Rather, all possible combinations can be used over the remaining 7 bits of each byte, producing a *dense* encoding. This is the key to improve the compression ratio achieved by Tagged Huffman Code, which has to avoid some values of these 7 bits in each byte, since such values are prefixes of other codewords (remember that the tag bit of THC is not enough to produce a prefix code, and hence a Huffman coding over the remaining 7 bits is mandatory in order to maintain a prefix code). Thus, ETDC yields a better compression ratio than Tagged Huffman Code while keeping all its good searching and decompression capabilities. On the other hand, ETDC is easier to build and faster in both compression and decompression.

Example 1 Assume we have a text with a vocabulary of ten words and that we compress it with target symbols of three bits. Observe in Table 1 the fourth most frequent symbol. Using THC, the target symbol 111 cannot be used as a codeword by itself since it is a prefix of other codewords. However, ETDC can use symbol 111 as a codeword, since it cannot be a prefix of any other codeword due to the flag bit. The same happens with the seventh most frequent word in THC: The target symbols 111 011 cannot be used as a codeword, as again they are reserved as a prefix of other codewords.

□

In general, ETDC can be defined over target symbols of b bits, although in this paper we focus on the byte-oriented version where $b = 8$. ETDC is formally defined as follows.

Definition 1 *The b-ary End-Tagged Dense Code assigns to the i-th most frequent source symbol (starting*

with $i = 0$), a codeword of k digits in base 2^b, where

$$2^{b-1}\ \frac{2^{(b-1)(k-1)}-1}{2^{b-1}-1}\ \leq\ i\ <\ 2^{b-1}\ \frac{2^{(b-1)k}-1}{2^{b-1}-1}.$$

Those k digits are filled with the representation of number $i - 2^{b-1}\ \frac{2^{(b-1)(k-1)}-1}{2^{b-1}-1}$ in base 2^{b-1} (most to least significant digit), and we add 2^{b-1} to the least significant digit (that is, the last digit).

That is, for $b = 8$, the first word ($i = 0$) is encoded as $\underline{1}0000000$, the second as $\underline{1}0000001$, until the 128^{th} as $\underline{1}1111111$. The 129^{th} word is encoded as $\underline{0}0000000$:$\underline{1}0000000$, 130^{th} as $\underline{0}0000000$:$\underline{1}0000001$ and so on until the $(128^2 + 128)^{th}$ word $\underline{0}1111111$:$\underline{1}1111111$.

The number of words encoded with 1, 2, 3, etc., bytes is fixed (specifically 128, 128^2, 128^3 and so on). Definition 1 gives the formula for the change points in codeword lengths (ranks $i = 2^{b-1}\ \frac{2^{(b-1)(k-1)}-1}{2^{b-1}-1}$).

Note that the code depends on the rank of the words, not on their actual frequency. That is, if we have four words *A, B, C, D* (ranked 1 to 4) with frequencies 0.36, 0.22, 0.22, and 0.20, respectively, then the code will be the same as if their frequencies were 0.9, 0.09, 0.009, and 0.001. As a result, only the sorted vocabulary must be stored with the compressed text for the decompressor to rebuild the model. Therefore, the vocabulary will be basically of the same size as in the case of Huffman codes, yet Huffman codes need some extra information about the shape of the Huffman tree (which is nevertheless negligible using canonical Huffman trees).

As it can be seen in Table 2, the computation of the code is extremely simple: It is only necessary to sort the source symbols by decreasing frequency and then sequentially assign the codewords taking care of the flag bit. This permits the coding phase to be faster than using Huffman, as obtaining the codewords is simpler.

On the other hand, it is also easy to assign the codeword of an isolated rank i. Following Definition 1, it is easy to see that we can encode a rank and decode a codeword in $O((\log i)/b)$ time. Section 5 presents those algorithms.

Word rank	codeword assigned	# Bytes	# words
0	**1**0000000	1	
...	...	...	
$2^7 - 1 = 127$	**1**1111111	1	
$2^7 = 128$	**0**0000000:**1**0000000	2	
...	...	...	
256	**0**0000001:**1**0000000	2	$2^7 2^7$
...	...	...	
$2^7 2^7 + 2^7 - 1 = 16511$	**0**1111111:**1**1111111	2	
$2^7 2^7 + 2^7 = 16512$	**0**0000000:**0**0000000:**1**0000000	3	
...	...	...	$(2^7)^3$
$(2^7)^3 + (2^7)^2 + 2^7 - 1$	**0**1111111:**0**1111111:**1**1111111	3	
...	...	...	

Table 2: Code assignment in End-Tagged Dense Code.

Actually, the idea of ETDC is not new if we see it under a different light. What we are doing is to encode the symbol frequency rank with a variable-length integer representation. The well-known *universal codes* C_α, C_γ and C_ω (Elias 1975) also assign codewords to source symbols in order of decreasing probability, with shorter codewords for the first positions. Other authors proposed other codes with similar characteristics (Lakshmanan 1981, Fraenkel and Klein 1996). These codes yield an average codeword length within a constant factor of the optimal average length. Unfortunately, the constant may be too large for the code to be preferable over one based on the probabilities such as Huffman, which is optimal but needs to know the distribution in advance. The reason is that these codes adapt well only to some probability distributions,

which may be far away from those of our interest. More specifically, C_α is suitable when the distribution is very skewed (more than our vocabularies), while C_γ and C_ω behave better when no word is much more likely than any other. These codes do not adjust well to large and moderately skewed vocabularies, as those of text databases. Moreover, we show in Section 4 how ETDC can be adapted better to specific vocabulary distributions.

It is possible to bound the compression performance of ETDC in terms of the text entropy or in terms of Huffman performance. Let E_b be the average codeword length, measured in target symbols[3], using a b-ary ETDC (that is, using target symbols of b bits), and H_b the same using a b-ary Huffman code. As ETDC is a prefix code and Huffman is the optimal prefix code, we have $H_b \leq E_b$. On the other hand, as ETDC uses all the combinations on $b-1$ bits (leaving the other for the flag), its codeword is shorter than H_{b-1}, where sequences that are prefixes of others are forbidden. Thus $H_b \leq E_b \leq H_{b-1}$. The average length of a Huffman codeword is smaller than one target symbol over the zero-order entropy of the text (Bell et al. 1990). Let $\mathcal{H}$ be the zero-order entropy measured in bits. Thus, $\mathcal{H} \leq bH_b < \mathcal{H} + b$, and the same holds for H_{b-1}. We conclude that

$$\frac{\mathcal{H}}{b} \;\leq\; E_b \;<\; \frac{\mathcal{H}}{b-1} + 1.$$

While the first inequality is obvious, the second tells us that the average number of bits used by a b-ary ETDC is at most $\frac{b}{b-1}\,\mathcal{H} + b$. It also means that $E_b < \frac{b}{b-1}\,H_b + 1$, which upper bounds the coding inefficiency of ETDC with respect to a b-ary Huffman. Several studies about bounds on Dense Codes and b-ary Huffman codes applied to Zipf (Zipf 1949) and Zipf-Mandelbrot (Mandelbrot 1953) distributions can be found in (Navarro and Brisaboa 2006, Fariña 2005).

As shown in Section 6, ETDC improves Tagged Huffman Code compression ratio by more than 8%. Its difference with respect to Plain Huffman Code is just around 2.5%, much less than the rough upper bound just obtained. On the other hand, the encoding time with ETDC is just 40% below that of Plain Huffman Code, and one can search ETDC as fast as Tagged Huffman Code.

4 (s, c)-Dense Code

Instead of thinking in terms of tag bits, End-Tagged Dense Code can be seen as using 2^{b-1} values, from 0 to $2^{b-1} - 1$, for the symbols that do not end a codeword, and using the other 2^{b-1} values, from 2^{b-1} to $2^b - 1$, for the last symbol of the codewords. Let us call *continuers* the former values and *stoppers* the latter. The question that arises now is whether that proportion between the number c of continuers and s of stoppers is optimal. That is, for a given text collection with a specific word frequency distribution, we want to use the optimal number of continuers and stoppers. Those will probably be different from $s = c = 2^{b-1}$. Thus (s, c)-Dense Code is a generalization of ETDC, where any $s + c = 2^b$ can be used (in particular, the values maximizing compression). ETDC is actually a $(2^{b-1}, 2^{b-1})$-Dense Code.

This idea has been previously pointed out by Rautio et al. (2002). They presented an encoding scheme using stoppers and continuers on a character-based source alphabet, yet their goal is to have a code where searches can be efficiently performed. Their idea is to create a code where each codeword can be split into two parts in such a way that searches can be performed using only one part of the codewords.

Example 2 illustrates the advantages of using a variable rather than a fixed number of stoppers and continuers.

Example 2 Assume that 5,000 distinct words compose the vocabulary of the text to compress, and that $b = 8$ (byte-oriented code).

If End-Tagged Dense Code is used, that is, if the number of stoppers and continuers is $2^7 = 128$, there will be 128 codewords of one byte, and the rest of the words would have codewords of two bytes, since

[3]For example, if binary Huffman is used (b=2), the target alphabet will be $\Sigma = \{0, 1\}$, while if it is byte oriented ($b = 8$), the target alphabet will be $\Sigma = \{0, \ldots, 255\}$.

$128 + 128^2 = 16,512$. That is, 16,512 is the number of words that can be encoded with codewords of one or two bytes. Therefore, there would be $16,512 - 5,000 = 11,512$ unused codewords of two bytes.

If the number of stoppers chosen is 230 (so the number of continuers is $256 - 230 = 26$), then $230 + 230 \times 26 = 6,210$ words can be encoded with codewords of only one or two bytes. Therefore all the 5,000 words can be assigned codewords of one or two bytes in the following way: the 230 most frequent words are assigned one-byte codewords and the remaining $5,000 - 230 = 4,770$ words are assigned two-byte codewords.

It can be seen that words from 1 to 128 and words ranked from 231 to 5,000 are assigned codewords of the same length in both schemes. However words from 129 to 230 are assigned to shorter codewords when using 230 stoppers instead of only 128.

□

This shows that it can be advantageous to adapt the number of stoppers and continuers to the size and the word frequency distribution of the vocabulary.

4.1 Formalization

In this section, we formally define (s, c)-Dense Code and prove some of its properties. Formally, this section contains the material of Section 3, yet we have chosen to present ETDC first because it is more intuitively derived from the previous Tagged Huffman Code. We start by defining (s, c) stop-cont codes as follows.

Definition 2 *Given positive integers s and c, a (s, c) stop-cont code assigns to each source symbol a unique target code formed by a sequence of zero or more digits in base c (that is, from 0 to $c - 1$), terminated with a digit between c and $c + s - 1$.*

It should be clear that a stop-cont coding is just a base-c numerical representation, with the exception that the last digit is between c and $c + s - 1$. Continuers are digits between 0 and $c - 1$ and stoppers are digits between c and $c + s - 1$. The next property clearly follows.

Property 1 Any (s, c) stop-cont code is a prefix code.

Proof: If one codeword were a prefix of the other, since the shorter codeword must have a final digit of value not smaller than c, then the longer codeword should have an intermediate digit which is not in base c. This is a contradiction. □

Among all possible (s, c) stop-cont codes for a given probability distribution, (s, c)-Dense Code minimizes the average codeword length.

Definition 3 *Given positive integers s and c, (s, c)-Dense Code ((s, c)-DC, or SCDC) is a (s, c) stop-cont code that assigns the i-th most frequent source symbol (starting with $i = 0$) to a codeword of k digits in base $s + c$ (most significant digits first), where*

$$s\,\frac{c^{k-1}-1}{c-1} \;\leq\; i \;<\; s\frac{c^k-1}{c-1}$$

Let $x = i - s\,\frac{c^{k-1}-1}{c-1}$. Then, the first $k-1$ digits are filled with the representation of number $\lfloor x/s \rfloor$ in base c, and the last digit is $c + (x \bmod s)$.

To fix ideas, using bytes as symbols ($s + c = 2^8$), the encoding process can be described as follows:

- One-byte codewords from c to $c + s - 1$ are given to the first s words in the vocabulary.

- Words ranked from s to $s + sc - 1$ are assigned sequentially to two-byte codewords. The first byte of each codeword has a value in the range $[0, c - 1]$ and the second in the range $[c, c + s - 1]$.
- Words from $s + sc$ to $s + sc + sc^2 - 1$ are assigned to three-byte codewords, and so on.

Word rank	codeword assigned	# Bytes	# words
0	$[c]$	1	
...	...	...	s
$s-1$	$[c+s-1]$	1	
s	$[0][c]$	2	
...	...	...	
$s+s-1$	$[0][c+s-1]$	2	sc
$s+s$	$[1][c]$	2	
...	...	...	
$s+sc-1$	$[c-1][c+s-1]$	2	
$s+sc$	$[0][0][c]$	3	
...	...	...	sc^2
$s+sc+sc^2-1$	$[c-1][c-1][c+s-1]$	3	
...	...	...	

Table 3: Code assignment in (s, c)-Dense Code.

Table 3 summarizes this process. Next, we give an example of how codewords are assigned.

Example 3 The codewords assigned to twenty-two source symbols by a (2,6)-Dense Code are the following (from most to least frequent symbol): ⟨6⟩, ⟨7⟩, ⟨0,6⟩, ⟨0,7⟩, ⟨1,6⟩, ⟨1,7⟩, ⟨2,6⟩, ⟨2,7⟩, ⟨3,6⟩, ⟨3,7⟩, ⟨4,6⟩, ⟨4,7⟩, ⟨5,6⟩, ⟨5,7⟩ ⟨0,0,6⟩, ⟨0,0,7⟩, ⟨0,1,6⟩, ⟨0,1,7⟩, ⟨0,2,6⟩, ⟨0,2,7⟩, ⟨0,3,6⟩, ⟨0,3,7⟩. □

Notice that the code does not depend on the exact symbol probabilities, but only on their ordering by frequency. We now prove that the dense coding is an optimal stop-cont coding.

Property 2 The average length of a (s, c)-Dense Code is minimal with respect to any other (s, c) stop-cont code.

Proof: Let us consider an arbitrary (s, c) stop-cont code, and let us write all the possible codewords in numerical order, as in Table 3, together with the source symbol they encode, if any. Then it is clear that (i) any unused codeword in the middle could be used to represent the source symbol with longest codeword, hence a compact assignment of target symbols is optimal; and (ii) if a less probable source symbol with a shorter codeword is swapped with a more probable symbol with a longer codeword then the average codeword length decreases, and hence sorting the source symbols by decreasing frequency is optimal. □

It is now clear from Definition 3 that ETDC is a $(2^{b-1},2^{b-1})$-DC, and therefore (s, c)-DC is a generalization of ETDC, where s and c can be adjusted to optimize the compression for the distribution of frequencies and the size of the vocabulary. Moreover, $(2^{b-1},2^{b-1})$-DC (i.e. ETDC) is more efficient than Tagged Huffman Code over b bits, because Tagged Huffman Code is essentially a *non-dense* $(2^{b-1},2^{b-1})$ stop-cont code, while ETDC is a $(2^{b-1},2^{b-1})$-Dense Code.

Example 4 Table 4 shows the codewords assigned to a small set of words ordered by frequency when using Plain Huffman Code, (6,2)-DC; End-Tagged Dense Code (which is a (4,4)-DC), and Tagged Huffman Code. Digits of three bits (instead of bytes) are used for simplicity ($b = 3$), and therefore $s + c = 8$. The last four columns present the products of the number of bytes by the frequency of each word, and its sum (the average codeword length) is shown in the last row.

Rank	Word	Freq	PHC	(6,2)-DC	ETDC	THC	Freq × bytes			
							PHC	(6,2)-DC	ETDC	THC
0	A	0.200	[0]	[2]	[4]	[4]	0.20	0.20	0.20	0.20
1	B	0.200	[1]	[3]	[5]	[5]	0.20	0.20	0.20	0.20
2	C	0.150	[2]	[4]	[6]	[6]	0.15	0.15	0.15	0.15
3	D	0.150	[3]	[5]	[7]	[7][0]	0.15	0.15	0.15	0.30
4	E	0.140	[4]	[6]	[0][4]	[7][1]	0.14	0.14	0.28	0.28
5	F	0.090	[5]	[7]	[0][5]	[7][2]	0.09	0.09	0.18	0.18
6	G	0.040	[6]	[0][2]	[0][6]	[7][3][0]	0.04	0.08	0.08	0.12
7	H	0.020	[7][0]	[0][3]	[0][7]	[7][3][1]	0.04	0.04	0.04	0.06
8	I	0.005	[7][1]	[0][4]	[1][4]	[7][3][2]	0.01	0.01	0.01	0.015
9	J	0.005	[7][2]	[0][5]	[1][5]	[7][3][3]	0.01	0.01	0.01	0.015
					average codeword length		**1.03**	**1.07**	**1.30**	**1.52**

Table 4: Comparative example among compression methods, for b=3.

It is easy to see that, for this example, Plain Huffman Code and the (6,2)-Dense Code are better than the (4,4)-Dense Code (ETDC) and they are also better than Tagged Huffman Code. A (6,2)-Dense Code is better than a (4,4)-Dense Code because it takes advantage of the distribution of frequencies and of the number of words in the vocabulary. However the values (6,2) for s and c are not the optimal ones since a (7,1)-Dense Code obtains, in this example, an optimal compressed text having the same result as Plain Huffman Code. □

The problem now consists of finding the s and c values (assuming a fixed b where $2^b = s + c$) that minimize the size of the compressed text for a specific word frequency distribution.

4.2 Optimal s and c values

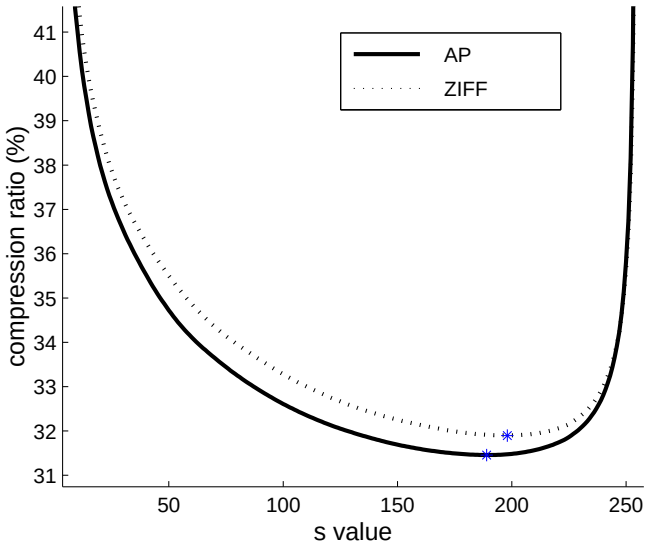
The key advantage of SCDC with respect to ETDC is the ability to use the optimal s and c values. In all the real text corpora used in our experiments (with $s + c = 2^8$), the size of the compressed text, as a function of s, has only one local minimum. Figure 1 shows compression ratios and file sizes as a function of s for two real corpora, ZIFF and AP[4]. It can be seen that a unique minimum exists for each collection, at $s = 198$ in ZIFF and $s = 189$ in AP. The table details the sizes and compression ratios when values of s close to the optimum are used.

The behavior of the curves is explained as follows. When s is very small, the number of high frequency words encoded with one byte is also very small (s words are encoded with one byte), but in this case c is large and therefore many words with low frequency will be encoded with few bytes: sc words will be encoded with two bytes, sc^2 words with 3 bytes, sc^3 with 4 bytes, and so on. From that point, as s grows, we gain compression in more frequent words and lose compression in less frequent words. At some later point, the compression lost in the last words is larger than the compression gained in words at the beginning, and therefore the global compression ratio worsens. That point gives us the optimal s value. Moreover, Figure 1 shows that, in practice, the compression is relatively insensitive to the exact value of s around the optimal value.

If one assumes the existence of a unique minimum, a binary search strategy permits computing the best s and c values in $O(b)$ steps. At each step, we check whether we are in the decreasing or increasing part of the curve and move towards the decreasing direction. However, the property of the existence of a unique minimum does not always hold. An artificial distribution with two local minima is given in Example 5 [5].

[4] These are two text collections from TREC-2, described in Section 6.

[5] This contradicts our earlier assertion (Brisaboa et al. 2003b), where we mentioned the existence of a long and complex



(a)

s value	AP Corpus		ZIFF Corpus	
	ratio(%)	size(bytes)	ratio(%)	size(bytes)
186	31.5927	79,207,309	32.0433	59,350,593
187	31.5917	79,204,745	32.0391	59,342,810
188	31.5910	79,202,982	32.0355	59,336,069
189	**31.5906**	**79,202,053**	32.0321	59,329,848
190	31.5907	79,202,351	32.0290	59,324,149
191	31.5912	79,203,574	32.0263	59,319,106
192	31.5921	79,205,733	32.0239	59,314,667
195	31.5974	79,219,144	32.0188	59,305,267
196	31.6002	79,226,218	32.0179	59,303,599
197	31.6036	79,234,671	32.0174	59,302,677
198	31.6074	79,244,243	**32.0174**	**59,302,609**
199	31.6117	79,254,929	32.0178	59,303,324
200	31.6166	79,267,125	32.0186	59,304,790
201	31.6220	79,280,683	32.0198	59,307,119

(b)

Figure 1: Compressed text sizes and compression ratios for different s values.

Example 5 Consider a text with $N = 50,000$ words, and $n = 5,000$ distinct words. An artificial distribution of the probability of occurrence p_i for all words $0 \leq i < n$ in the text is defined as follows:

$$p_i = \begin{cases} 0.4014 & \text{if } i = 0 \\ 0.044 & \text{if } i \in [1..9] \\ 0.0001 & \text{if } i \in [10..59] \\ 0.00004 & \text{if } i \in [60..4999] \end{cases}$$

If the text is compressed using (s, c)-Dense Code and assuming that $b = 4$ (therefore, $s + c = 2^b = 16$), the distribution of the size of the compressed text depending on the value of s used to encode words has two local minima, at $s = c = 8$ and at $s = 10,\ c = 6$. This situation can be observed in Table 5. □

Therefore, a binary search does not always guarantee that we will find the optimum s value, and a sequential search over the 2^b possible values is necessary. In practice, however, all the text collections we have considered do have a unique minimum and thus permit binary searching for the optimum s value.

proof of the existence of a unique minimum, which was finally flawed.

s value	avg. codeword length	compressed size (bytes)
7	1.70632	85,316
8	1.67716	**83,858**
9	1.67920	83,960
10	1.67900	**83,950**
11	1.71758	85,879

Table 5: Size of compressed text for an artificial distribution.

Section 4.3 explains why this property is expected to hold in all real-life text collections.

Let us consider how to find the optimum s value, either by binary or sequential search. Assume $s+c=2^b$ and $c>1$ in the following. As sc^{k-1} different source symbols can be encoded using k digits, let us call

$$W_k^s \;=\; \sum_{j=1}^{k} sc^{j-1} \;=\; s\,\frac{c^k-1}{c-1} \tag{1}$$

(where $W_0^s=0$) the number of source symbols that can be encoded with up to k digits. Let us consider an alphabet of n source symbols, where p_i is the probability of the i-th most frequent symbol ($i=0$ for the first symbol) and $p_i=0$ if $i\geq n$. Let us also define

$$P_j \;=\; \sum_{i=0}^{j} p_i$$

as the sum of p_i probabilities up to j. We also need

$$f_k^s \;=\; \sum_{i=W_{k-1}^s}^{W_k^s-1} p_i \;=\; P_{W_k^s-1} - P_{W_{k-1}^s}$$

the sum of probabilities of source symbols encoded with k digits by (s,c)-DC and

$$F_k^s \;=\; \sum_{i=0}^{W_k^s-1} p_i \;=\; P_{W_k^s-1}$$

the sum of probabilities of source symbols encoded with up to k digits by (s,c)-DC. Then, the average codeword length, $L(s,c)$, for (s,c)-DC is

$$\begin{aligned} L(s,c) \;&=\; \sum_{k=1}^{K^s} k\; f_k^s \;=\; \sum_{k=1}^{K^s} k\;(F_k^s - F_{k-1}^s) \;=\; \sum_{k=0}^{K^s-1} (k-(k+1))\;F_k^s \;+\; K^s F_{K^s}^s \\ &=\; K^s - \sum_{k=0}^{K^s-1} F_k^s \;=\; \sum_{k=0}^{K^s-1} (1-F_k^s) \end{aligned} \tag{2}$$

where $K^s=\left\lceil \log_c\left(1+\frac{n(c-1)}{s}\right)\right\rceil$ is the maximum codeword length used by the code, and thus $F_{K^s}^s=1$.

Thus, if we precompute all the sums P_j in $O(n)$ time, we can compute $L(s,c)$ in $O(K^s)=O(\log_c n)$ time. Therefore, a binary search requires $O(n+b\log n)$ and a sequential search requires $O(n+2^b\log n)$ time. We can assume $2^b\leq n$, as otherwise all the n codewords fit in one symbol. This makes the binary search complexity $O(n)$, while the sequential search complexity can be as large as $O(n\log n)$ if b is close to $\log n$.

The case $c=1$ is hardly useful in practice: $W_k^s=ks$, $K^s=\lceil n/s\rceil$, and $L(s,c)$ is computed in $O(n)$ time.

s	W_1^s	W_2^s	W_3^s	W_4^s	W_5^s	W_6^s
1	1	256	65,281	16,711,681	4.E+09	1.E+17
10	10	2,470	607,630	150,082,150	4.E+10	9.E+17
20	20	4,740	1,118,660	265,117,700	6.E+10	1.E+18
30	30	6,810	1,539,090	349,366,650	8.E+10	2.E+18
40	40	8,680	1,874,920	406,849,000	9.E+10	**2.E+18**
50	50	10,350	2,132,150	441,344,750	**9.E+10**	2.E+18
60	60	11,820	2,316,780	**456,393,900**	9.E+10	2.E+18
70	70	13,090	2,434,810	455,296,450	8.E+10	2.E+18
80	80	14,160	2,492,240	441,112,400	8.E+10	1.E+18
90	90	15,030	**2,495,070**	416,661,750	7.E+10	1.E+18
100	100	15,700	2,449,300	384,524,500	6.E+10	9.E+17
110	110	16,170	2,360,930	347,040,650	5.E+10	7.E+17
120	120	16,440	2,235,960	306,310,200	4.E+10	6.E+17
127	127	16,510	2,129,917	276,872,827	4.E+10	5.E+17
128	128	**16,512**	2,113,664	272,646,272	3.E+10	4.E+17
129	129	16,512	2,097,153	268,419,201	3.E+10	4.E+17
130	130	16,510	2,080,390	264,193,150	3.E+10	4.E+17
140	140	16,380	1,900,220	222,309,500	3.E+10	3.E+17
150	150	16,050	1,701,450	182,039,250	2.E+10	2.E+17
160	160	15,520	1,490,080	144,522,400	1.E+10	1.E+17
170	170	14,790	1,272,110	110,658,950	1.E+10	8.E+16
180	180	13,860	1,053,540	81,108,900	6.E+09	5.E+16
190	190	12,730	840,370	56,292,250	4.E+09	2.E+16
200	200	11,400	638,600	36,389,000	2.E+09	1.E+16
210	210	9,870	454,230	21,339,150	1.E+09	5.E+10
220	220	8,140	293,260	10,842,700	390,907,660	1.E+10
230	230	6,210	161,690	4,359,650	113,662,090	3.E+09
240	240	4,080	65,520	1,110,000	17,883,120	286,314,480
250	250	1,750	10,750	73,750	460,750	2,791,750
255	**255**	510	765	1,275	2,040	3,060

Table 6: Values of W_k^s for $1 \leq k \leq 6$.

4.3 Uniqueness of the minimum

As explained, only one minimum appeared in all our experiments with $b = 8$ and real collections. In this section, we show that this is actually expected to be the case in all practical situations. We remark that we are not going to give a proof, but just intuitive arguments of what is expected to happen in practice.

Heaps Law (Heaps 1978) establishes that $n = O(N^\beta)$, where n is the vocabulary size, N the number of words in the text, and β a constant that depends on the text type and is usually between 0.4 and 0.6 (Baeza-Yates and Ribeiro-Neto 1999). This means that the vocabulary grows very slowly in large text collections. In fact, in a corpus obtained by joining different texts from TREC-2 and TREC-4 (see Section 6), which adds up to 1 gigabyte, we find only 886,190 different words. This behavior, combined with the fact that we use bytes ($b = 8$) as target symbols, is the key to the uniqueness of the minimum in practice.

Table 6 shows how the W_k^s (Eq. (1)) evolve, with $b = 8$. The maximum number of words that can be encoded with 6 bytes is found when the value of s is around 40. In the same way, the maximum number of words that can be encoded with 5, 4, and 3 bytes is reached when the value of s is respectively around of 50, 60 and 90. Finally, the value of s that maximizes the number of words encoded with 2 bytes is $s = c = 128$, but the number of words encoded with just one byte grows when s is increased.

Notice that compression clearly improves, even if a huge vocabulary of 2 million words is considered, when s increases from $s = 1$ until $s = 128$. This is because, up to $s = 128$, the number of words that can be encoded with 1 byte and with 2 bytes both grow, while the number that can be encoded with 3 bytes stays larger than 2 million. Only larger vocabularies can lose compression if s grows from $s = 90$ (where 2.5 million words can be encoded) up to $s = 128$. This happens because those words that can be encoded with 3-byte codewords for $s = 90$, would need 4-byte codewords when s increases. However, as it has been already pointed out, we never obtained a vocabulary with more than $886,190$ words in all the real texts used, and that number of words is encoded with just 3 bytes with any $s \leq 187$.

Therefore, in our experiments (and in most real-life collections) the space trade-off depends on the sum of the probability of the words encoded with only 1 byte, against the sum of the probability of words encoded with 2 bytes. The remaining words are always encoded with 3 bytes.

The average codeword length for two consecutive values of s are (Eq. (2))

$$\begin{aligned} L(s,c) &= 1 + \sum_{i \geq W_1^s} p_i + \sum_{i \geq W_2^s} p_i, \\ L(s+1,c-1) &= 1 + \sum_{i \geq W_1^{s+1}} p_i + \sum_{i \geq W_2^{s+1}} p_i. \end{aligned}$$

Two different situations arise depending on whether $s > c$ or $s \leq c$. When $s < c$ the length $L(s,c)$ is always greater than $L(s+1,c-1)$ because the number of words that are encoded with both 1 and 2 bytes grows when s increases. Therefore, as the value of s is increased, compression improves until the value $s = c = 128$ is reached. For s values beyond $s = c$ ($s > c$), compression improves when the value $s+1$ is used instead of s iff $L(s+1,c-1) < L(s,c)$, that is,

$$\sum_{i=W_1^s}^{W_1^{s+1}-1} p_i > \sum_{i=W_2^{s+1}}^{W_2^s-1} p_i, \quad \text{which is} \quad p_s > \sum_{i=sc+c}^{s+sc-1} p_i.$$

For each successive value of s, p_s clearly decreases. On the other hand, $\sum_{sc+c}^{sc+s-1} p_i$ grows. To see this, since the p_i values are decreasing, it is sufficient to show that each interval $[W_2^{s+1},\ W_2^s - 1]$ is longer than the former and it ends before the former starts (so it contains more and higher p_i values). That is: (a) $W_2^s - W_2^{s+1} > W_2^{s-1} - W_2^s$, and ($b$) $W_2^{s+1} < W_2^s$. It is a matter of algebra to see that both hold when $s \geq c$. As a consequence, once s reaches a value such that $p_s \leq \sum_{i=W_2^{s+1}}^{W_2^s-1} p_i$, successive values of s will also produce a loss of compression. Such loss of compression will increase in each successive step. Hence only one local minimum will exist.

The argument above is valid until s is so large that more than 3 bytes are necessary for the codewords, even with moderate vocabularies. For example, for $s = 230$ we will need more than 3 bytes for n as low as 161,690, which is perfectly realistic. When this happens, we have to take into account limits of the form W_3^s and more, which do not have the properties (a) and (b) of W_2^s. Yet, notice that, as we move from s to $s+1$, compression improves by p_s (which decreases with s) and it deteriorates because of two facts: (1) fewer words are coded with 2 bytes; (2) fewer words are coded with k bytes, for any $k \geq 3$. While factor (2) can grow or reduce with s, factor (1) always grows because of properties (a) and (b) satisfied by W_2^s. Soon the losses because of factor (1) are so large that the possible oscillations due to factor (2) are completely negligible, and thus local minima do not appear anyway.

To illustrate the magnitudes we are considering, assume the text satisfies Zipf (1949) Law with $\theta = 1.5$, which is far lower than the values obtained in practice (Baeza-Yates and Ribeiro-Neto 1999). If we move from $s = 230$ to $s + 1 = 231$, compression gain is $p_s < 0.00029$. The loss just because of W_2^s is > 0.00042, and the other W_k^s ($k \geq 3$) add almost other 0.00026 to the loss. So, no matter what happens with factor (2), factor (1) is strong enough to ensure that compression will deteriorate for $s > 230$.

5 Encoding, decoding, and searching

Encoding, decoding, and searching is extremely simple in ETDC and SCDC. We give in this section the algorithms for general SCDC. The case of ETDC can be considered as a particular case of SCDC taking the value of s as 2^{b-1} (128 in our case). We will make clear where the particular case of ETDC yields potentially more efficiency.

5.1 Encoding algorithm

Encoding is usually done in a sequential fashion as shown in Table 3, in time proportional to the output size. Alternatively, on-the-fly encoding of individual words is also possible. Given a word rank i, its k-byte codeword can be easily computed in $O(k) = O(\log_c i)$ time[6].

There is no need to store the codewords (in any form such as a tree) nor the frequencies in the compressed file. It is enough to store the plain words sorted by frequency and the value of s used in the compression process. Therefore, the vocabulary will be basically of the same size as using Huffman codes and canonical trees.

Figure 2 gives the sequential encoding algorithm. It computes the codewords for all the words in the sorted vocabulary and stores them in a vector *code*.

```
SequentialEncode (code, n, s, c)
(1)  firstKBytes ← 0;   // rank of the first word encoded with k bytes
(2)  numKBytes ← s;   // number of k-byte codewords
(3)  p ← 0;   // current codeword being generated
(4)  k ← 1;   // size of the current codeword
(5)  while p < n   // n codewords are generated
(6)      paux ← 0;   //relative position among k-byte codewords
(7)      while (p < n) and (paux < numKBytes) // k-byte codewords are computed
(8)          code[p].codeword[k − 1] ← c + (paux mod s); // rightmost byte
(9)          paux ← paux div s;
(10)         for i ← k − 2 downto 0
(11)             code[p].codeword[i] ← paux mod c;
(12)             paux ← paux div c;
(13)         p ← p + 1;
(14)         paux ← p − firstKBytes;
(15)     k ← k + 1;
(16)     firstKBytes ← firstKBytes + numKBytes;
(17)     numKBytes ← numKBytes × c;
```

Figure 2: Sequential encoding process. It receives the vocabulary size n, and the (s, c) parameters, and leaves the codewords in *code*.

Notice that, when using ETDC, since $s = c = 128$, the operations *mod* s, *mod* c, *div* s, *div* c, $+ c$ and $\times c$ can be carried out using faster bitwise operations. As shown in Section 6.2.1, this yield better encoding times.

Figure 3 presents the on-the-fly encoding algorithm. It outputs the bytes of the codeword one at a time from right to left, that is, the least significant bytes first.

5.2 Decoding algorithm

The first step of decompression is to load the words that compose the vocabulary to a vector. Those are already sorted by frequency.

In order to obtain the word that corresponds to a given codeword, the decoder runs a simple computation to obtain, from the codeword, the rank of the word i. Then it outputs the i-th vocabulary word. Figure 4 shows the decoding algorithm. It receives a codeword x, and iterates over each byte of x. The end of the codeword can be easily recognized because its value is not smaller than c. After the iteration, the value i holds the relative rank of the word among those of k bytes. Then $base = W^s_{k-1}$ is added to obtain the absolute rank. Overall, a codeword of k bytes can be decoded in $O(k) = O(\log_c i)$ time[7].

[6] If $c = 1$, encoding takes $O(i/2^b)$ time.

[7] If $c = 1$, decoding takes $O(i/2^b)$ time.

```
Encode (i, s, c)
(1)  output c + (i mod s);
(2)  x ← i div s;
(3)  while x > 0
(4)       x ← x − 1;
(5)       output x mod c;
(6)       x ← x div c;
```

Figure 3: On-the-fly encoding process. It receives the word rank i, and the (s, c) parameters, and outputs the codewords in reverse order.

```
Decode (x, s, c)
(1)  i ← 0;
(2)  base ← 0; // W^s_{k−1}
(3)  tot ← s; // sc^{k−1}
(4)  k ← 1; // number of bytes of the codeword
(5)  while x[k] < c
(6)       i ← i × c + x[k];
(7)       k ← k + 1;
(8)       base ← base + tot;
(9)       tot ← tot × c;
(10) i ← i × s + (x[k] − c);
(11) return i + base;
```

Figure 4: On-the-fly decoding process. It receives a codeword x, and the (s, c) parameters, and returns the word rank. It is also possible to have *base* precomputed for each codeword size.

5.3 Searching

The essential idea of searching a semistatic code is to search the compressed text for the compressed pattern P. However, some care must be exercised to make sure that the occurrences of codeword P only correspond to whole text codewords. Specifically, we must ensure that the following situations do not occur: (i) P matches the prefix of a text codeword A; (ii) P matches the suffix of a text codeword A; (iii) P matches strictly within a text codeword A; (iv) P matches within the concatenation of two codewords $A : B$ or more.

Case (i) cannot occur in prefix codes (such as Huffman or our Dense Codes). However, cases (ii) to (iv) can occur in Plain Huffman Code. This is why searches on Plain Huffman Code must proceed byte-wise so as to keep track of codeword beginnings. Tagged Huffman Code, on the other hand, rules out the possibility of situations (ii) to (iv) thanks to the flag bit that distinguishes codeword beginnings. Hence Tagged Huffman Code can be searched without any care using any string matching algorithm.

End-Tagged Dense Code, on the other hand, uses the flag bit to signal the end of a codeword. (s, c)-Dense Code does not have such a tag, yet a value $\geq c$ serves anyway to distinguish the end of a codeword. It is easy to see that situations (iii) and (iv) cannot arise in this case, yet case (ii) is perfectly possible. This is because Dense Codes are not *suffix codes*, that is, a codeword can be a suffix of another.

Yet, with a small modification, we can still use any string matching algorithm (in particular, the Boyer-Moore family, which permits skipping text bytes). We can just run the search algorithm without any care and, each time a matching of the whole pattern P occurs in the text, we check whether the occurrence corresponds to a whole text codeword or to just a suffix of a text codeword. For this sake, it is sufficient to check whether the byte preceding the first matched byte is $\geq c$ or not. Figure 5 shows an example of how false matchings can be detected (using "bytes" of three bits and ETDC). Note that Plain Huffman Code

does not permit such simple checking.

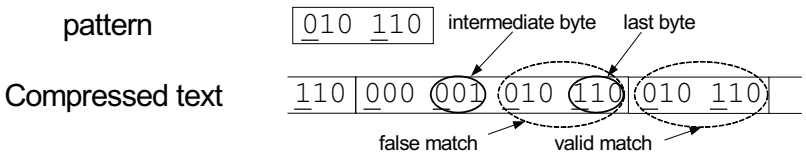


Figure 5: Searching using End-Tagged Dense Code.

This overhead in searches is negligible because checking the previous byte is only needed when a matching is detected, which is infrequent. As shown in Section 6.4, this small disadvantage with respect to Tagged Huffman Code (which is both a prefix and a suffix code) is compensated because the size of the compressed text is smaller with Dense Codes than with Tagged Huffman Code.

Figure 6 gives a search algorithm based on Horspool's variant of Boyer-Moore (Horspool 1980, Navarro and Raffinot 2002). This algorithm is especially well suited to this case (codewords of length at most 3–4, characters with relatively uniform distribution in $\{0 \ldots 255\}$).

6 Empirical results

We present in this section experimental results on compression ratios, as well as speed in compression, decompression, and searching. We used some text collections from TREC-2[8]: AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF); and from TREC-4: Congressional Record 1993 (CR) and Financial Times 1991 to 1994 (FT91 to FT94). In addition, we included the small Calgary corpus[9] (CALGARY), and two larger collections: ALL_FT aggregates corpora FT91 to FT94, and ALL aggregates all our corpora.

We compressed the corpora with Plain Huffman Code (PHC), Tagged Huffman Code (THC), our End-Tagged Dense Code (ETDC) and our (s, c)-Dense Code (SCDC), using in all cases bytes as the target symbols ($b = 8$). In all cases, we defined words as maximal contiguous sequences of letters and digits, we distinguished upper and lower case, and we adhered to the spaceless word model (Moura et al. 2000). Under this model, words and separators share a unique vocabulary, and single-space separators are assumed by default and thus not encoded (i.e., two contiguous codewords representing words imply a single space between them).

[8] `http://trec.nist.gov`.

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

```
Search (x, k, c, T, z)
(1)  for b ← 0 to 255 do d[b] ← k
(2)  for j ← 1 to k − 1 do d[x[j]] ← k − j
(3)  i ← 0
(4)  while i ≤ z − k
(5)      j ← k − 1
(6)      while j ≥ 0 and T[i + j] = x[j] do j ← j − 1
(7)      if j < 0 and (i = 0 or T[i − 1] ≥ c) output i
(8)      i ← i + d[T[i + k − 1]]
```

Figure 6: Search process using Horspool's algorithm. It receives a codeword x and its length k, parameter c, the compressed text T to search, and its length z. It outputs all the text positions that start a true occurrence of x in T.

We also include comparisons against the most competitive classical compressors. These turn out to be adaptive techniques that do not permit direct access nor competitive direct searching, even compared to PHC. These are Gnu *gzip* [10], a Ziv-Lempel compressor (Ziv and Lempel 1977); Seward's *bzip2* [11], a compressor based on the Burrows-Wheeler transform (Burrows and Wheeler 1994); and Moffat's *arith* [12], a zero-order word-based modeler coupled with an arithmetic coder (Carpinelli et al. 1999). *Gzip* and *bzip2* have options `-f` where they run faster, and `-b`, where they compress more.

6.1 Compression ratio

Table 7 shows the compression ratios obtained by the four semistatic techniques when compressing the different corpora. The second column gives the original size in bytes. Columns 3 to 6 (sorted by compression performance) give the compression ratios for each method. Column 7 shows the (small) loss of compression of SCDC with respect to PHC, and the last column shows the difference between SCDC and ETDC. The fourth column, which refers to SCDC, gives also the optimal (s, c) values.

We excluded the size of the compressed vocabulary in the results. This size is negligible and similar in all cases, although a bit smaller in SCDC and ETDC because only the sorted words are needed.

Corpus	Original Size	PHC	SCDC	ETDC	THC	$1 - \frac{PHC}{SCDC}$ (%)	$1 - \frac{SCDC}{ETDC}$ (%)
CALGARY	2,131,045	34.76	(197,59) 35.13	35.92	38.91	1.06	2.30
FT91	14,749,355	30.26	(193,63) 30.50	31.15	33.58	0.79	2.09
CR	51,085,545	29.21	(195,61) 29.45	30.10	32.43	0.81	2.16
FT92	175,449,235	30.49	(193,63) 30.71	31.31	34.08	0.72	1.92
ZIFF	185,220,215	31.83	(198,58) 32.02	32.72	35.33	0.59	2.14
FT93	197,586,294	30.61	(195,61) 30.81	31.49	34.30	0.65	2.16
FT94	203,783,923	30.57	(195,61) 30.77	31.46	34.28	0.65	2.19
AP	250,714,271	31.32	(189,67) 31.59	32.14	34.72	0.85	1.71
ALL_FT	591,568,807	30.73	(196,60) 30.88	31.56	34.16	0.49	2.15
ALL	1,080,719,883	32.05	(188,68) 32.24	32.88	35.60	0.59	1.95

Table 7: Comparison of compression ratio among semistatic compressors. Vocabulary sizes are excluded.

PHC obtains the best compression ratio (as expected from an optimal prefix code). ETDC always obtains better results than THC, with an improvement of 7.7%–9.0%. SCDC improves ETDC compression ratio by 1.7%–2.3%, and it is worse than the optimal PHC only by 0.49%–1.06% (being 0.6% in the whole corpus ALL). Therefore, our Dense Codes retain the good searchability and random access of THC, but their compression ratios are much closer to those of the optimum PHC.

Table 8 compares ETDC and SCDC against adaptive compressors. This time we have added the size of the vocabulary, which was compressed with classical (character oriented, bit-based) Huffman. It can be seen that SCDC always compresses more than *gzip -b*, except on the very small Calgary file. The improvement in compression ratio is between 7% and 14% except on the ZIFF collection. The other compressors, however, compress more than SCDC by a margin of 6% to 24% (even excluding the short Calgary file). We remind that these compressors do not permit local decompression nor direct search. We also show next that they are very slow at decompression.

[10] `http://www.gnu.org`.
[11] `http://www.bzip.org`.
[12] `http://www.cs.mu.oz.au/~alistair/arith_coder/`.

Corpus	Original Size	SCDC	ETDC	*arith*	*gzip -f*	*gzip -b*	*bzip2 -f*	*bzip2 -b*
CALGARY	2,131,045	43.64	44.43	34.68	43.53	36.840	32.83	28.92
FT91	14,749,355	33.50	34.15	28.33	42.57	36.33	32.31	27.06
CR	51,085,545	30.80	31.45	26.30	39.51	33.18	29.51	24.14
FT92	175,449,235	31.68	32.28	29.82	42.59	36.38	32.37	27.09
ZIFF	185,220,215	32.92	33.62	26.36	39.66	32.98	29.64	25.11
FT93	197,586,294	31.68	32.36	27.89	40.20	34.12	30.62	25.32
FT94	203,783,923	31.54	32.23	27.86	40.24	34.12	30.54	25.27
AP	250,714,271	32.14	32.69	28.00	43.65	37.23	33.26	27.22
ALL_FT	591,568,807	31.45	32.13	27.85	40.99	34.85	31.15	25.87
ALL	1,080,719,883	32.72	33.36	27.98	41.31	35.00	31.30	25.98

Table 8: Comparison of compression ratio against adaptive compressors.

6.2 Compression time

In this section we compare the Dense and Plain Huffman encoding phases and measure the code generation time. We do not include THC because it works exactly as PHC, with the only difference of generating 2^{b-1}-ary trees instead of 2^b-ary trees as PHC. This causes some loss in encoding speed. We also include the adaptive compressors.

The model used for compressing a corpus in our experiments is described in Figure 7. It consists of three phases.

1. *Vocabulary extraction.* The corpus is processed once in order to obtain all distinct words in it, and their number of occurrences (n). The result is a list of pairs $(word, frequency)$, which is then sorted by $frequency$. This phase is identical for PHC, SCDC, and ETDC.

2. *Encoding.* Each vocabulary word is assigned a codeword. This process is different for each method:

   - The PHC encoding phase is the application of the Huffman technique (Moffat and Katajainen 1995, Moffat and Turpin 1996, Huffman 1952). Encoding takes $O(n)$ time overall.
   - The SCDC encoding phase has two parts: The first computes the list of accumulated frequencies and searches for the optimal s and c values. Its cost is $O(n)$ in practice (Sections 4.2 and 4.3). After obtaining the optimal s and c values, sequential encoding is performed (Figure 2). The overall cost is $O(n)$.
   - The encoding phase is even simpler in ETDC than in SCDC, because ETDC does not have to search for the optimal s and c values (as they are fixed to 128). Therefore only the sequential code generation phase is performed. It costs $O(n)$ time overall.

   In all cases, the result of the encoding section is a *hash table* of pairs (*word*, *codeword*).

3. *Compression.* The whole source text is processed again. For each source word, the compression process looks for it in the hash table and outputs its corresponding codeword.

Note that both PHC and SCDC encoding phases run in linear time. However, Huffman's constant is in practice larger because it involves more operations than just adding up frequencies.

Given that the vocabulary extraction phase, the process of building the hash table of pairs, and the compression phase are identical in PHC, ETDC and SCDC, we first measure only code generation time ($T_1 - T_0$ in Figure 7), to appreciate the speedup due to the simplicity of Dense Codes. We then measure also the overall compression time, to assess the impact of encoding in the whole process and to compare against adaptive compressors.

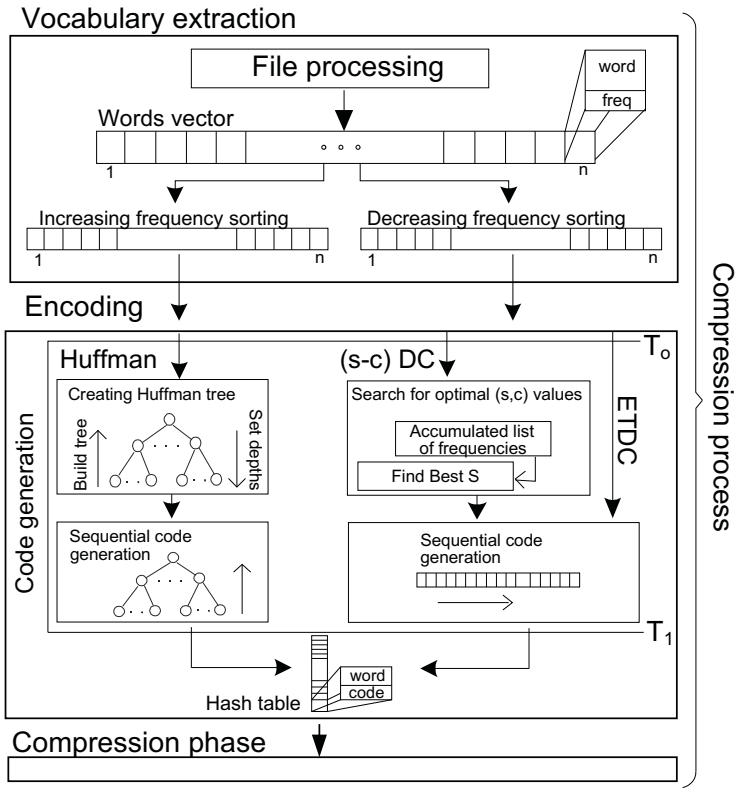


Figure 7: Vocabulary extraction and encoding phases.

6.2.1 Encoding time

Table 9 shows the results on code generation time. A dual Intel®pentium®-III 800 Mhz system, with 768 SDRAM-100Mhz was used in our tests. It ran Debian GNU/Linux (kernel version 2.2.19). The compiler was `gcc` version 2.95.2 with *-O9* compiler optimizations. The results measure encoding user time.

The second column gives the corpus size measured in words, while the third gives the number of different words (vocabulary size). Columns 4–6 give encoding time in milliseconds for the different techniques (from faster to slower). Column 7 measures the gain of ETDC over SCDC, and the last the gain of SCDC over PHC.

ETDC takes advantage of its simpler encoding phase with respect to SCDC, to reduce its encoding time by about 25%. This difference corresponds to two factors. One is that, when using ETDC, s is 128, and then bitwise operations (as shown in Section 5.1) can be used. Another factor is the amount of time needed to compute the optimal s and c values, which corresponds mainly to the process of computing the vector of accumulated frequencies. With respect to PHC, ETDC decreases the encoding time by about 60%. Code generation is always about 45% faster for SCDC than for PHC.

Although the encoding time is in all methods $O(n)$ under reasonable conditions, ETDC and SCDC perform simpler operations. Computing the list of accumulated frequencies and searching for the best (s, c) pair only involve elementary operations, while the process of building a canonical Huffman tree has to deal with the tree structure.

Corpus	Size (words)	n	ETDC (msec)	SCDC (msec)	PHC (msec)	$1 - \frac{ETDC}{SCDC}$ (%)	$1 - \frac{SCDC}{PHC}$ (%)
CALGARY	528,611	30,995	4.233	6.150	11.133	31.165	44.760
FT91	3,135,383	75,681	11.977	15.350	26.500	21.976	42.075
CR	10,230,907	117,713	21.053	25.750	49.833	18.239	48.328
FT92	36,803,204	291,427	52.397	69.000	129.817	24.063	46.848
ZIFF	40,866,492	284,892	44.373	56.650	105.900	21.671	46.506
FT93	42,063,804	295,018	52.813	69.725	133.350	24.255	47.713
FT94	43,335,126	269,141	52.980	71.600	134.367	26.006	46.713
AP	53,349,620	237,622	50.073	64.700	121.900	22.607	46.924
ALL_FT	124,971,944	577,352	103.727	142.875	260.800	27.400	45.217
ALL	229,596,845	886,190	165.417	216.225	402.875	23.498	46.330

Table 9: Code generation time comparison.

6.2.2 Overall compression time

Section 6.2.1 only shows the encoding time, that is, the time to assign a codeword to each word in the sorted vocabulary. We now consider the whole compression time. Table 10 shows the compression times in seconds for the four techniques and for the adaptive compressors.

Corpus	PHC	SCDC	ETDC	THC	*arith*	*gzip -f*	*gzip -b*	*bzip2 -f*	*bzip2 -b*
CALGARY	0.415	0.405	0.393	0.415	1.030	0.360	1.095	2.180	2.660
FT91	2.500	2.493	2.482	2.503	6.347	2.720	7.065	14.380	18.200
CR	7.990	7.956	7.988	8.040	21.930	8.875	25.155	48.210	65.170
FT92	29.243	29.339	29.230	29.436	80.390	34.465	84.955	166.310	221.460
ZIFF	30.354	30.620	30.368	30.725	82.720	33.550	82.470	174.670	233.250
FT93	32.915	33.031	32.783	33.203	91.057	36.805	93.135	181.720	237.750
FT94	33.874	33.717	33.763	33.700	93.467	37.500	96.115	185.107	255.220
AP	42.641	42.676	42.357	42.663	116.983	50.330	124.775	231.785	310.620
ALL_FT	99.889	100.570	100.469	101.471	274.310	117.255	293.565	558.530	718.250
ALL	191.396	191.809	191.763	192.353	509.710	188.310	532.645	996.530	1,342.430

Table 10: Compression time comparison (in seconds).

Observe that the two passes over the original file (one for vocabulary extraction and the other to compress the file) take the same time with all semistatic techniques. These tasks dominate the overall compression time, and blur out most of the differences in encoding time. The differences are in all cases very small, around 1%.

It is interesting that the time to write the compressed file benefits from a more compact encoding, as it has to write less data. Therefore, a technique with fastest encoding like ETDC is harmed in the overall time for its extra space with respect to SCDC and PHC.

As a result, PHC is faster in the largest files, since it has to write a shorter output file. However, in shorter files, where the size of the output file is not so different and the vocabulary size is more significant, ETDC is faster due to its faster encoding time. SCDC is in between and it is the fastest in some intermediate size files. THC, on the other hand, is consistently the slowest, as it is slow at encoding and produces the largest output files.

With respect to adaptive compressors, we can see that all PHC, THC, ETDC and SCDC, are usually 10%-20% faster than *gzip* (even considering its fastest mode), and 2.5–6.0 times faster than *bzip2* and *arith* compression (only those two defeat ETDC and SCDC compression ratios).

6.3 Decompression time

Decoding is almost identical for PHC, THC, ETDC and SCDC. Such a process starts by loading the words of the vocabulary into a vector V. For decoding a codeword, SCDC needs the s value used in compression, and PHC and THC also need to load two vectors: *base* and $first$ (Moffat and Katajainen 1995). Next, the compressed text is read and each codeword is replaced by its corresponding uncompressed word. Since it is possible to detect the end of a codeword by using the s value in the case of SCDC (128 when ETDC is used) and the $first$ vector in the case of PHC and THC, decompression is performed codeword-wise. Given a codeword C, a simple decoding algorithm obtains the position i such that $V[i]$ is the uncompressed word that corresponds to codeword C. The decompression process takes $O(z)$ time (where z is the size of the compressed text).

Table 11 gives overall decompression time in seconds for the four semistatic techniques and for adaptive compressors. Again, in addition to the simplicity of the decoding process, we must consider the penalty posed by codes with worse compression ratios, as they have to process a longer input file.

Corpus	PHC	SCDC	ETDC	THC	*arith*	*gzip -f*	*gzip -b*	*bzip2 -f*	*bzip2 -b*
CALGARY	0.088	0.097	0.085	0.092	0.973	0.090	0.110	0.775	0.830
FT91	0.577	0.603	0.570	0.575	5.527	0.900	0.825	4.655	5.890
CR	1.903	1.971	1.926	1.968	18.053	3.010	2.425	15.910	19.890
FT92	7.773	7.592	7.561	7.801	65.680	8.735	7.390	57.815	71.050
ZIFF	8.263	7.988	7.953	8.081	67.120	9.070	8.020	58.790	72.340
FT93	8.406	8.437	8.694	8.657	71.233	10.040	9.345	62.565	77.860
FT94	8.636	8.690	8.463	8.825	75.925	10.845	10.020	62.795	80.370
AP	11.040	11.404	11.233	11.637	88.823	15.990	13.200	81.875	103.010
ALL_FT	24.798	25.118	24.500	26.280	214.180	36.295	30.430	189.905	235.370
ALL	45.699	46.698	46.352	47.156	394.067	62.485	56.510	328.240	432.390

Table 11: Decompression time comparison (in seconds).

We can see again that the times for PHC, THC, ETDC, and SCDC are very similar. The differences are under 1.5 seconds even when decompressing 1 gigabyte corpora. It is nevertheless interesting to comment upon some aspects. PHC benefits from the smaller size of the compressed file. ETDC processes a larger file, yet it is faster than PHC in many cases thanks to its simpler and faster decoding. SCDC is between both in compression ratio and decoding cost, and this is reflected in its overall decompression time, where it is never the fastest but it is intermediate in many cases. THC is again the slowest method, as its input file is the largest and its decoding algorithm is as slow as that of PHC.

These four compressors, on the other hand, are usually 10%–20% faster than *gzip* (even using its fastest mode), and 7–9 times faster than *bzip2* and *arith*.

6.4 Search time

As explained in Section 5.3, we can search on ETDC and SCDC using any string matching algorithm provided we check every occurrence to determine whether it is a valid match. The check consists of inspecting the byte preceding the occurrence (Figure 6).

In this section, we aim at determining whether this check influences search time or not, and in general which is the performance of searching ETDC and SCDC compared to searching THC. Given a search pattern, we find its codeword and then search the compressed text for it, using Horspool's algorithm (Horspool 1980). This algorithm is depicted in Figure 6, where it is adapted to ETDC and SCDC. In the case of searching THC, line (7) is just "**if** $j < 0$ **output** i".

Comparing methods that encode the same word in different ways is not simple. The time needed by Horspool's algorithm is inversely proportional to the length of the codeword sought. If one method encodes a word with 2 bytes and the other with 3 bytes, Horspool will be much faster for the second method.

In our first experiment, we aim at determining the relative speed of searching ETDC, SCDC, and THC, when codewords of the same length are sought (classical compressors are not competitive at searching and hence left out of this experiment). In order to isolate the length factor, we carefully chose the most and least frequent words that are assigned 1, 2, and 3 byte codewords with the three methods simultaneously. ETDC and SCDC never had codewords longer than 3 bytes, although THC has. We consider later the issue of searching for "random" codewords.

Table 12 compares the search times in the ALL corpus. The first three columns give the length of the codeword, the most and least frequent words chosen for that codeword length, and the number of occurrences of those words in the text. The frequency of the words is important because ETDC and SCDC must check every occurrence, but THC must not. Columns 4–7 give the search times in seconds for the three techniques (fastest to slowest). The last two columns give the decrease in search time of SCDC with respect to ETDC and THC.

code length	word	occurrences	SCDC (sec)	ETDC (sec)	THC (sec)	$1 - \frac{SCDC}{ETDC}$ (%)	$1 - \frac{SCDC}{THC}$ (%)
1	`the`	8,205,778	5.308	5.452	5.759	2.636	7.826
1	`were`	356,144	4.701	5.048	5.182	6.856	9.273
2	`sales`	88,442	2.526	2.679	2.800	5.711	9.779
2	`predecessor`	2,775	2.520	2.666	2.736	5.476	7.895
3	`resilience`	612	1.671	1.779	1.858	6.061	10.059
3	`behooves`	22	1.613	1.667	1.701	3.245	5.189

Table 12: Searching time comparison.

It can be seen that searching ETDC for a k-bytes codeword is 2%–5% faster than searching THC. Even when searching for `"the"`, which has one true occurrence every 42 bytes on average, the search in ETDC is faster. This shows that the extra comparison needed in ETDC is more than compensated by its better compression ratio (which implies that a shorter file has to be traversed during searches). The same search algorithm gives even better results in SCDC, because the compression ratio is significantly better. Searching SCDC is 2.5%–7% faster than searching ETDC, and 5%–10% faster than searching THC.

Let us now consider the search for random words. As expected from Zipf (1949) Law, a few vocabulary words account for 50% of the text words (those are usually *stopwords*: articles, prepositions, etc.), and a large fraction of vocabulary words appear just a few times in the text collection. Thus picking the words from the text and from the vocabulary is very different. The real question is which is the distribution of searched words in text databases. For example, stopwords are never searched for in isolation, because they bring a huge number of occurrences and their information value is null (Baeza-Yates and Ribeiro-Neto 1999). On the other hand, many words that appear only once in our text collections are also irrelevant as they are misspellings or meaningless strings, thus they will not be searched for either.

It is known that the search frequency of vocabulary words follows a Zipf distribution as well, which is not related to that of the word occurrences in the text (Baeza-Yates and Navarro 2004). As a rough approximation, we have followed the model (Moura et al. 2000) where each vocabulary word is sought with uniform probability. Yet, we have discarded words that appear only once, trying to better approximate reality. As the search and occurrence distributions are independent, this random-vocabulary model is reasonable.

The paradox is that this model (and reality as well) *favors* the worst compressors. For example, THC has many 4-byte codewords in our largest text collections, whereas ETDC and SCDC use at most 3 bytes.

On large corpora, where THC uses 4 bytes on a significant part of the vocabulary, there is a good chance of picking a longer codeword with THC than with ETDC and SCDC. As a result, the Horspool search for that codeword will be faster on THC.

More formally, the Horspool search for a codeword of length m in a text of N words, for which we achieved c bytes/word compression, costs essentially Nc/m byte comparisons. Calling m_{THC} and m_{SCDC} the average codeword lengths (of words randomly chosen from the vocabulary) in both methods, and c_{THC} and c_{SCDC} the bytes/word compressions achieved (that is, average codeword length in the text), then searching THC costs Nc_{THC}/m_{THC} and searching SCDC costs Nc_{SCDC}/m_{SCDC}. The ratio of search time of THC divided by SCDC is $\frac{c_{THC}/c_{SCDC}}{m_{THC}/m_{SCDC}}$. Because of Zipf Law, the ratio m_{THC}/m_{SCDC} might be larger than c_{THC}/c_{SCDC}, and thus searching THC can be faster than searching SCDC.

Table 13 shows the results of searching for (the same set of) 10,000 random vocabulary words, giving mean and standard deviation for each method.

Corpus	SCDC		ETDC		THC	
	$\overline{\text{time}}$	σ	$\overline{\text{time}}$	σ	$\overline{\text{time}}$	σ
FT91	0.023	0.006	0.024	0.006	0.024	0.005
CR	0.072	0.015	0.073	0.016	0.077	0.012
FT92	0.250	0.038	0.257	0.044	0.267	0.046
ZIFF	0.275	0.041	0.283	0.047	0.292	0.052
FT93	0.283	0.049	0.291	0.045	0.299	0.052
FT94	0.291	0.039	0.300	0.047	0.306	0.059
AP	0.376	0.056	0.382	0.066	0.380	0.048
ALL_FT	0.844	0.091	0.867	0.101	0.760	0.141
ALL	1.610	0.176	1.650	0.195	1.390	0.250

Table 13: Searching for random patterns.

The results are as expected after the discussion. Searching THC is up to 13.7% faster on very long text collections (ALL_FT and ALL), thanks to the many 4-byte codeword assignments it makes. On the other hand, SCDC is 5.0%–6.5% faster than THC on medium-size text collections. Even ETDC is 2%-5% faster than THC on those texts.

The next experiment compares multi-pattern searching on a text compressed with ETDC and SCDC against those searches on uncompressed text. To search the compressed text we applied the *Set Horspool's* algorithm (Horspool 1980, Navarro and Raffinot 2002), with the small modifications needed to deal with our codes. Three different algorithms were tested to search the uncompressed text: *i)* our own implementation of the Set Horspool's algorithm, *ii)* author's implementation of *Set Backward Oracle Matching algorithm* (SBOM) (Allauzen et al. 1999), and *iii)* the *agrep* software (Wu and Manber 1992b, Wu and Manber 1992a), a fast approximate pattern-matching tool which allows, among other things, searching a text for multiple patterns. Agrep searches the text and returns those chunks containing one or more searched patterns. The default chunk is a line, as the default chunk-separator is the newline character. Once the first search pattern is found in a chunk, agrep skips processing the remaining bytes in the chunk. This speeds up agrep searches when a large number of patterns are searched. However, it does not give the exact positions of all the searched patterns. To make a fairer comparison, in our experiments, we also tried agrep with the *reversed* patterns, which are unlikely to be found. This maintains essentially the same statistics of the searched patterns and reflects better the real search cost of agrep.

By default, the search tools compared in our experiments (except agrep) run in *silent* mode, and count the number of occurrences of the patterns in the text. Agrep was forced to use these two options by setting the parameters *-s -c*.

Search type	lenght of pattern	number of patterns							
		5	10	25	50	100	200	400	1000
	5	0.665	0.784	1.136	1.764	2.493	3.211	3.473	3.836
ETDC	10	0.665	0.800	1.147	1.754	2.469	3.152	3.389	3.703
Set Horspool	>10	0.655	0.769	1.139	1.729	2.458	3.167	3.388	3.655
	5	0.621	0.713	1.023	1.552	2.294	3.228	3.585	3.940
SCDC	10	0.630	0.717	1.005	1.542	2.278	3.172	3.648	3.898
Set-Horspool	>10	0.611	0.711	0.992	1.496	2.248	3.151	3.585	3.893
	5	2.099	3.077	5.333	7.286	9.010	10.942	12.989	16.314
Set-Horspool	10	1.864	2.913	4.546	5.533	6.648	7.959	9.409	12.482
	>10	1.759	2.811	4.285	5.255	6.148	7.253	8.665	11.726
	5	4.230	5.220	6.552	8.351	11.040	13.608	16.557	22.895
SBOM	10	2.970	3.532	4.995	6.401	7.572	9.363	12.024	17.197
	>10	2.757	3.458	4.875	6.043	7.250	8.946	11.705	17.162
	5	5.410	5.100	4.520	3.686	2.200	1.150	0.531	0.218
Agrep -s -c	10	2.910	3.150	3.700	3.855	3.310	2.330	1.568	0.717
default	>10	2.690	2.960	3.580	3.792	3.390	2.880	1.985	1.070
	5	5.720	5.867	6.257	7.205	10.723	9.862	8.490	5.464
Agrep -s -c	10	2.861	3.035	3.532	4.614	4.758	5.028	5.857	6.462
rev. patterns	>10	2.617	2.817	3.357	4.167	4.311	4.579	5.038	6.152

Table 14: Multi-pattern searches over the ALL corpora (in seconds).

Table 14 shows the average time (in seconds) needed to search for 1000 sets of random words. To choose these sets of random words we considered the influence of the length of the words in searches (a longer pattern is usually searched for faster) by separately considering words of length 5, 10, or greater than 10. With respect to the number of words in each set, we considered several values from 5 to 1000.

The results clearly show that searching compressed text is much faster than searching the uncompressed version of the text. Only *default* agrep (which skips lines where patterns are found) can improve the results of compressed searches, yet this occurs when more than 100 words are searched for and, as explained, does not reflect the real cost of a search. In the case of agrep with *reversed patterns*, it is interesting to distinguish two cases: *i) Searching for inverted patterns of length greater or equal than* 10. Those inverted patterns do never occur in the text. Therefore, search time worsens as the number of searched patterns increases. *ii) Searching for inverted patterns whose length is* 5. In this case, some of the inverted patterns are found in the text, and the probability of finding the searched patterns increases as the number of searched patterns grows. This fact, explains that search time improves when the number of search patterns is large (> 100 patterns).

If we focus on our implementations of the Set Horspool's algorithm applied to both compressed and plain text (which is the fairest comparison between searches in compressed and plain text), we see that searching compressed text is around 3-5 times faster than searching plain text.

It is also noticeable that compressed search time does not depend on the length of the uncompressed pattern (in fact, the small differences shown are related to the number of occurrences of the searched patterns). On the other hand, searching plain text for longer patterns is faster, as it permits skipping more bytes during the traversal of the searched file.

Moura et al. (2000) showed that if searchers allowing errors are considered, then searching text compressed with THC (which has roughly the same search times as our Dense Codes) can be up to 8 times faster than searching the uncompressed version of the text.

We have left aside from this comparison the block-boundary method proposed by Moura et al. (2000) to permit Horspool searching on PHC. In the experiments they report, that method is 7% slower than THC on medium-size collections (where SCDC is at least 5% faster than THC), and the block-alignment poses a space overhead of 0.78% over PHC (where SCDC overhead is around 0.60%). Thus SCDC is even closer to the optimum PHC compression ratio and 12%–19% faster. On large corpora, the same codeword length effect that affects SCDC will make the search on PHC even slower, as those codewords are shorter on average.

7 Conclusions and future work

We have presented End-Tagged Dense Code (ETDC) and (s, c)-Dense Code (SCDC), two statistical semistatic word-based compression techniques suitable for text databases. SCDC is a generalization of ETDC which improves its compression ratio by adapting the number of stopper/continuer values to the corpus to be compressed.

Although the best semistatic compression ratio is achieved with Huffman coding, different variants that lose some compression in exchange for random access and fast searching for codewords are preferable in compressed text databases. In particular, Tagged Huffman Code (Moura et al. 2000) has all those features in exchange for 11% compression loss compared to Plain Huffman Code.

We have shown that ETDC and SCDC maintain all the good searching features of Tagged Huffman Code, yet their compression ratio is much closer to that of the optimum Plain Huffman Code (just 0.6% off). In addition, ETDC and SCDC are much simpler and faster to program, create, and manipulate.

Figure 8(a) summarizes compression ratio, encoding time, compression and decompression time, and search time for the semistatic word-based statistical methods. In the figure, the measures obtained in the AP corpus are shown normalized to the worst value. The lower the value in a bar, the better the result.

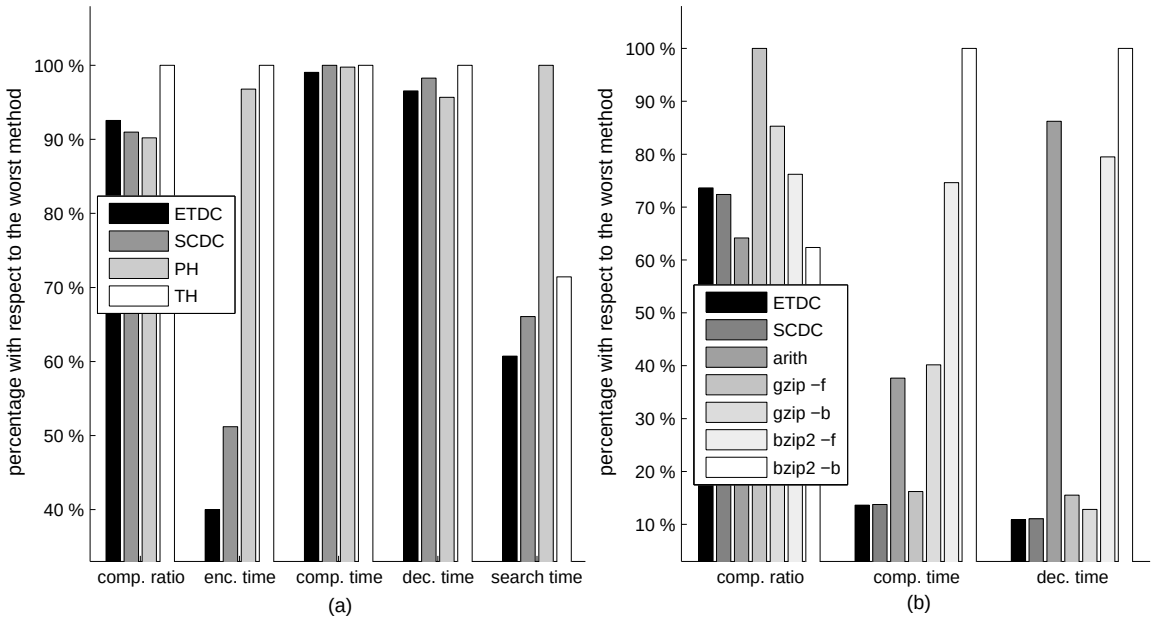


Figure 8: Comparison of Dense Codes against other compressors on corpus AP.

Using the same corpus and format, Figure 8(b) compares Dense Codes against other popular compressors: *gzip*, *bzip2*, and *arith*. We recall that *gzip* and *bzip2* have options "-f" (fast compression) and "-b" (best compression).

Figure 9 presents the comparison in terms of space/time tradeoffs for encoding and search time versus compression ratio (overall compression/decompression times are very similar in all cases). The figure illustrates that, while Plain Huffman Code remains interesting because it has the best compression ratio, Tagged Huffman Code has been overcome by both ETDC and SCDC in all concerns: compression ratio, encoding speed, compression and decompression speed, and search speed. In addition, Dense Codes are

simpler to program and manipulate. We also note that more complex searches (such as regular expression or approximate searching) can be handled with ETDC or SCDC just as with Plain Huffman Code (Moura et al. 2000), by means of a byte-wise processing of the text.

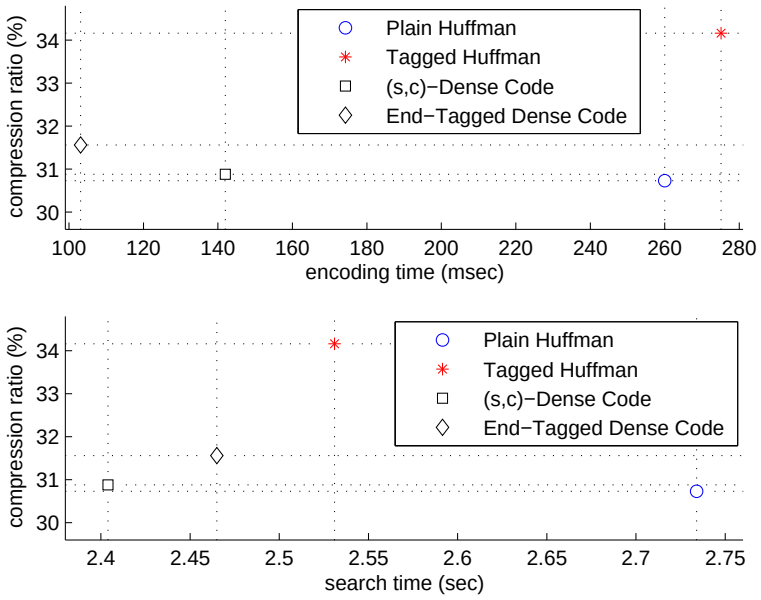


Figure 9: Space/time tradeoffs of Dense and Huffman-based codes on corpus AP.

Figure 10 compares ETDC and SCDC against other popular compressors in terms of space/time tradeoffs for compression and decompression versus compression ratio. It can be seen that *gzip* is overcome by our compressors in all aspects: ETDC and SCDC obtain better compression ratio, compression time, and decompression time, than *gzip*. Although *arith* and *bzip2* compress up to 20% more than ETDC and SCDC, these are 2.5–6 times faster to compress and 7–9 times faster to decompress. This shows that, even disregarding the fact that these classical compressors cannot perform efficient local decompression or direct text search, our new compressors are an attractive alternative even from a pure-compression point of view. We remark that these conclusions are only true if one is compressing not-too-small (say, at least 10 megabytes) natural language text collections.

ETDC and SCDC can be considered the first two members of a newborn family of *dense compressors*. Dense compressors have a number of possibilities in other applications. For example, we have faced the problem of dealing with growing text collections. Semi-static codes have an important drawback for compressed text databases: the whole corpus being compressed has to be available before compression starts. As a result, a compressed text database must periodically recompress all its text collections to accommodate new text that has been added, or accept a progressive degradation of compression ratio because of changes in the distribution of its collections. We have some preliminary results on a semistatic compressor based on Dense Codes (Brisaboa et al. 2005a). The idea is that, instead of changing any codeword assignment when a word increases its frequency, we concatenate its new occurrence with the next word and consider the pair as a new source symbol. Instead of using shorter codewords for more frequent symbols, we let codewords encode more source symbols when those appear. This is easy to do thanks to the simplicity of Dense Codes.

This leads naturally to the use of Dense Codes as *variable to variable* codes. This is a relatively new

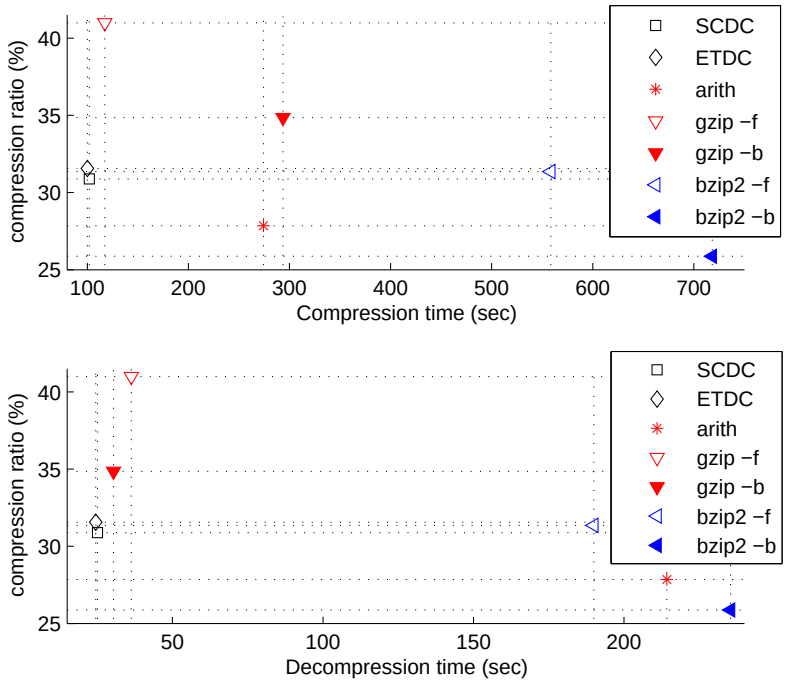


Figure 10: Space/time tradeoffs of Dense Codes versus adaptive compressors, on corpus AP.

research field (Savari and Szpankowski 2002), based on the use of source symbols of variable length, which are in turn encoded with variable-length codewords.

We have also explored the use of Dense Codes in dynamic scenarios (Brisaboa et al. 2004, Fariña 2005), where we have presented adaptive versions of both word-based byte-oriented Huffman and End-Tagged Dense Codes. These approaches do not permit efficient direct search on the compressed text since the assignment word/codeword varies frequently as the model changes during the compression.

The simplicity of Dense Codes, which results in much faster encoding time, has little impact in the overall compression time of semistatic compressors, as encoding is a tiny part of the whole process. In adaptive compression, however, updating the model upon each new word is a heavy task that must be carried out for every new text word. As a result, the overall compression time is much better with Dense Codes than with adaptive Huffman codes, whereas the compression ratio is almost the same.

Recently, we have managed to modify dynamic ETDC to allow direct search in the compressed text (Brisaboa et al. 2005b). This adaptive technique gives more stability to the codewords assigned to the original symbols as the compression progresses, exploiting the fact that Dense Codes depend only on the frequency rank and not the actual frequency of words. Basically, the technique changes the model only when it is absolutely necessary to maintain the original compression ratio, and it breaks the usual compressor-decompressor symmetry present in most adaptive compression schemes. This permits much lighter decompressors and searchers, which is very valuable in mobile applications.

Finally, Dense Codes have proved to be a valuable analytic tool to bound the redundancy of D-ary Huffman codes on different families of distributions (Navarro and Brisaboa 2006).

References

Allauzen, C., Crochemore, M. and Raffinot, M.: 1999, Factor oracle: A new structure for pattern matching., *SOFSEM*, LNCS 1725, pp. 295–310.

Baeza-Yates, R. and Navarro, G.: 2004, Recent advances in applied probability, *in* R. Baeza-Yates, J. Glaz, H. Gzyl, J. Husler and J. Palacios (eds), *Modeling Text Databases*, Springer, pp. 1–25.

Baeza-Yates, R. and Ribeiro-Neto, B.: 1999, *Modern Information Retrieval*, Addison-Wesley Longman.

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

Boyer, R. S. and Moore, J. S.: 1977, A fast string searching algorithm, *Communications of the ACM* **20**(10), 762–772.

Brisaboa, N., Fariña, A., Navarro, G. and Esteller, M.: 2003a, (s,c)-dense coding: An optimized compression code for natural language text databases, *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, LNCS 2857, Springer-Verlag, pp. 122–136.

Brisaboa, N., Fariña, A., Navarro, G. and Paramá, J.: 2004, Simple, fast, and efficient natural language adaptive compression, *Proceedings of the 11th International Symposium on String Processing and Information Retrieval (SPIRE'04)*, LNCS 3246, Springer–Verlag, pp. 230–241.

Brisaboa, N., Fariña, A., Navarro, G. and Paramá, J.: 2005a, Compressing dynamic text collections via phrase-based coding, *Proceedings of the 9th European Conference on Research and Advanced Technology for Digital Libraries (ECDL'05)*, LNCS 3652, Springer–Verlag, pp. 462–474.

Brisaboa, N., Fariña, A., Navarro, G. and Paramá, J.: 2005b, Efficiently decodable and searchable natural language adaptive compression, *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'05)*, ACM Press, pp. 234–241.

Brisaboa, N., Iglesias, E. L., Navarro, G. and Paramá, J. R.: 2003b, An efficient compression code for text databases, *Proceedings of the 25th European Conference on IR Research (ECIR'03)*, LNCS 2633, Springer–Verlag, pp. 468–481.

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

Carpinelli, J., Moffat, A., Neal, R., Salamonsen, W., Stuiver, L., Turpin, A. and Witten, I.: 1999, Word, character, integer, and bit based compression using arithmetic coding.
http://www.cs.mu.oz.au/~alistair/arith_coder/

Elias, P.: 1975, Universal codeword sets and the representation of the integers, *IEEE Transactions on Information Theory* **21**, 194–203.

Fariña, A.: 2005, *New Compression Codes for Text Databases*, PhD thesis, Database Laboratory, University of A Coruña.
http://coba.dc.fi.udc.es/~fari/phd/

Fraenkel and Klein: 1996, Robust universal complete codes for transmission and compression, *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science* **64**, 31–55.

Gage, P.: 1994, A new algorithm for data compression, *C Users Journal* **12**(2), 23–38.

Heaps, H. S.: 1978, *Information Retrieval: Computational and Theoretical Aspects*, Academic Press, New York.

Horspool, R. N.: 1980, Practical fast searching in strings, *Software Practice and Experience* **10**(6), 501–506.

Huffman, D. A.: 1952, A method for the construction of minimum redundancy codes, *Proceedings of the Institute of Electronics and Radio Engineers (IRE)* **40**(9), 1098–1101.

Klein, S. T. and Shapira, D.: 2005, Pattern matching in Huffman encoded texts, *Information Processing and Management* **41**(4), 829–841.

Lakshmanan, K. B.: 1981, On universal codeword sets., *IEEE Transactions on Information Theory* **27**(5), 659–662.

Manber, U.: 1997, A text compression scheme that allows fast searching directly in the compressed file, *ACM Transactions on Information Systems* **15**(2), 124–136.

Manber, U. and Wu, S.: 1994, GLIMPSE: A tool to search through entire file systems, *Proc. of the Winter 1994 USENIX Technical Conference*, pp. 23–32.

Mandelbrot, B.: 1953, An information theory of the statistical structure of language., *in* W. Jackson (ed.), *Communication Theory*, Academic Press N.Y., pp. 486–504.

Miyazaki, M., Fukamachi, S., Takeda, M. and Shinohara, T.: 1998, Speeding up the pattern matching machine for compressed texts, *Transactions of Information Processing Society of Japan* **39**(9), 2638–2648.

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

Moffat, A. and Katajainen, J.: 1995, In-place calculation of minimum-redundancy codes, *Proceedings of the 4th International Workshop on Algorithms and Data Structures (WADS'95)*, LNCS 955, pp. 393–402. Springer.

Moffat, A. and Turpin: 1996, On the implementation of minimum redundancy prefix codes, *IEEE Transactions on Communications* **45**, 170–179.

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)*, ACM Press, pp. 298–306.

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**(2), 113–139.

Navarro, G. and Brisaboa, N.: 2006, New bounds on D-ary optimal codes, *Information Processing Letters* **96**(5), 178–184.

Navarro, G., Moura, E., Neubert, M., Ziviani, N. and Baeza-Yates, R.: 2000, Adding compression to block addressing inverted indexes, *Information Retrieval* **3**(1), 49–77.

Navarro, G. and Raffinot, M.: 2002, *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press.

Navarro and Tarhio, J.: 2000, Boyer-Moore string matching over Ziv-Lempel compressed text, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848 in *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Montréal, Canada, pp. 166–180.

Navarro and Tarhio, J.: 2005, LZgrep: A Boyer-Moore String Matching Tool for Ziv-Lempel Compressed Text. *Software Practice and Experience (SPE)* **35**(12), 1107–1130.

Rautio, J., Tanninen, J. and Tarhio, J.: 2002, String matching with stopper encoding and code splitting, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, LNCS 2373, pp. 42–52. Springer.

Savari, S. A. and Szpankowski, W.: 2002, On the analysis of variable-to-variable length codes, *Proceedings of 2002 IEEE International Symposium on Information Theory (ISIT'02)*, p. 176. See also http://citeseer.ist.psu.edu/616808.html

Shibata, Y., Matsumoto, T., Takeda, M., Shinohara, A. and Arikawa, S.: 2000, A Boyer-Moore type algorithm for compressed pattern matching, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, LNCS 1848, Springer–Verlag, pp. 181–194.

Takeda, M., Shibata, Y., Matsumoto, T., Kida, T., Shinohara, A., Fukamachi, S., Shinohara, T. and Arikawa, S.: 2001, Speeding up string pattern matching by text compression: The dawn of a new era, *Transactions of Information Processing Society of Japan* **42**(3), 370–384.

Turpin, A. and Moffat, A.: 1997, Fast file search using text compression, *Proceedings of the 20th Australian Computer Science Conference*, pp. 1–8.

Wan, R.: 2003, *Browsing and Searching Compressed Documents*, PhD thesis, Department of Computer Science and Software Engineering, University of Melbourne, Australia. http://eprints.unimelb.edu.au/archive/00000484/

Witten, I. H., Moffat, A. and Bell, T. C.: 1999, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann Publishers, USA.

Wu, S. and Manber, U.: 1992a, Agrep – a fast approximate pattern-matching tool, *Proceedings USENIX Winter 1992 Technical Conference*, San Francisco, CA, pp. 153–162.

Wu, S. and Manber, U.: 1992b, Fast text searching allowing errors, *Communications of the ACM* **35**(10), 83–91.

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

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

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

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