An Efficient Compression Code
for Text Databases [*]

Nieves R. Brisaboa[1], Eva L. Iglesias[2], Gonzalo Navarro[3] and José R. Paramá[2]

[1] Database Lab., Univ. da Coruña, Facultade de Informática, Campus de Elviña s/n,
15071 A Coruña, Spain. {brisaboa,parama}@udc.es
[2] Computer Science Dept., Univ. de Vigo , Escola Superior de Enxeñería Informática,
Campus As Lagoas s/n, 32001, Ourense, Spain. eva@uvigo.es
[3] Dept. of Computer Science, Univ. de Chile, Blanco Encalada 2120, Santiago, Chile.
gnavarro@dcc.uchile.cl

Abstract. We present a new compression format for natural language texts, allowing both exact and approximate search without decompression. This new code –called End-Tagged Dense Code– has some advantages with respect to other compression techniques with similar features such as the Tagged Huffman Code of [Moura et al., ACM TOIS 2000]. Our compression method obtains (*i*) better compression ratios, (*ii*) a smaller and simpler vocabulary representation, and (*iii*) a simpler and faster encoding. At the same time, it retains the most interesting features of the method based on the Tagged Huffman Code, i.e., exact search for words and phrases directly on the compressed text using any known sequential pattern matching algorithm, efficient word-based approximate and extended searches without any decoding, and efficient decompression of arbitrary portions of the text. As a side effect, our analytical results give new upper and lower bounds for the redundancy of d-ary Huffman codes.

Keywords: Text compression, *D*-ary Huffman coding, text databases.

1   Introduction

Text compression techniques are based on exploiting redundancies in the text to represent it using less space [3]. The amount of text collections has grown in recent years mainly due to the widespread use of digital libraries, documental databases, office automation systems and the Web. Current text databases contain hundreds of gigabytes and the Web is measured in terabytes. Although the capacity of new devices to store data grows fast, while the associated costs decrease, the size of the text collections increases also rapidly. Moreover, CPU speed grows much faster than that of secondary memory devices and networks,

so storing data in compressed form reduces I/O time, which is more and more convenient even in exchange for some extra CPU time.

Therefore, compression techniques have become attractive methods to save space and transmission time. However, if the compression scheme does not allow to search for words directly on the compressed text, the retrieval will be less efficient due to the necessity of decompression before the search.

Classic compression techniques, as the well-known algorithms of Ziv and Lempel [15, 16] or the character oriented code of Huffman [4], are not suitable for large textual databases. One important disadvantage of these techniques is the inefficiency of searching for words directly on the compressed text. Compression schemes based on Huffman codes are not often used on natural language because of the poor compression ratios achieved. On the other hand, Ziv and Lempel algorithms obtain better compression ratios, but the search for a word on the compressed text is inefficient. Empirical results [10] showed that searching on a Ziv-Lempel compressed text can take half the time of decompressing that text and then searching it. However, the compressed search is twice as slow as just searching the uncompressed version of the text.

In [12], Moura et al. present a compression scheme that uses a semi-static word-based model and a Huffman code where the coding alphabet is byte-oriented. This compression scheme allows the search for a word on the compressed text without decompressing it in such a way that the search can be up to eight times faster for certain queries. The key idea of this work (and others like that of Moffat and Turpin [7]) is the consideration of the text words as the symbols that compose the text (and therefore the symbols that should be compressed). Since in Information Retrieval (IR) text words are the atoms of the search, these compression schemes are particularly suitable for IR. This idea has been carried on further up to a full integration between inverted indexes and word-based compression schemes, opening the door to a brand new family of low-overhead indexing methods for natural language texts [13, 8, 17].

Two basic search methods are proposed in [12]. One handles plain Huffman code (over words) and explores one byte of the compressed text at a time. This is quite efficient, but not as much as the second choice, which compresses the pattern and uses any classical string matching strategy, such as Boyer-Moore [9]. For this second, faster, choice to be of use, one has to ensure that no spurious occurrences are found. The problem is that a text occurrence of the code of a word may correspond to the concatenation of other codes instead of to the occurrence of the word. Although Plain Huffman Code is a prefix code (that is, no code is a prefix of the other), it does not ensure that the above problem cannot occur. Hence Moura et al. propose a so-called Tagged Huffman Code, where a bit of each byte in the codes is reserved to signal the beginning of a code. The price is an increase of approximately 8%–10% in the size of the compressed file.

In this paper we show that, although Plain Huffman Code gives the shortest possible output when a source symbol is always substituted by the same code, Tagged Huffman Code largely underutilizes the representation. We show that, by signaling the end instead of the beginning of a code, the rest of the bits

can be used in all their combinations and the code is still a prefix code. The resulting code, which we call End-Tagged Dense Code, becomes much closer to the compression obtained by the Plain Huffman Code. Not only this code retains the ability of being searchable with any string matching algorithm, but also it is extremely simple to build (it is not based on Huffman at all) and permits a more compact vocabulary representation. So the advantages over Tagged Huffman Code are (*i*) better compression ratios, (*ii*) same searching possibilities, (*iii*) simpler and smaller vocabulary representation, (*iv*) simpler and faster coding.

2 Related Work

Huffman is a well-known coding method [4]. The idea of Huffman coding is to compress the text by assigning shorter codes to more frequent symbols. It has been proven that Huffman algorithm obtains an optimal (i.e., shortest total length) *prefix code* for a given text.

A code is called a *prefix code* (or instantaneous code) if no codeword is a prefix of any other codeword. A prefix code can be decoded without reference to future codewords, since the end of a codeword is immediately recognizable.

2.1 Word-Based Huffman Compression

The traditional implementations of the Huffman code are character based, i.e., they adopt the characters as the symbols of the alphabet. A brilliant idea [6] uses the words in the text as the symbols to be compressed. This idea joins the requirements of compression algorithms and of IR systems, as words are the basic atoms for most IR systems. The basic point is that a text is much easier to compress when regarded as a sequence of words rather than characters.

In [12, 17], a compression scheme is presented that uses this strategy combined with a Huffman code. From a compression viewpoint, character-based Huffman methods are able to reduce English texts to approximately 60% of their original size, while word-based Huffman methods are able to reduce them to 25% of their original size, because the distribution of words is much more biased than the distribution of characters.

The compression schemes presented in [12, 17] use a semi-static model, that is, the encoder makes a first pass over the text to obtain the frequency of all the words in the text and then the text is coded in the second pass. During the coding phase, original symbols (words) are replaced by codewords. For each word in the text there is a unique codeword, whose length varies depending on the frequency of the word in the text. Using the Huffman algorithm, shorter codewords are assigned to more frequent words.

The set of codewords used to compress a text are arranged as a tree with edges labeled by bits, such that each path from the root to a leaf spells out a different code. Since this is a prefix code, no code is represented by an internal tree node. On the other hand, each tree leaf corresponds to a codeword that represents a different word of the text. For decompression purposes, the

corresponding original text word is attached to each leaf, and the whole tree is seen as a representation of the vocabulary of the text. Hence, the compressed file is formed by the compressed text plus this vocabulary representation. The Huffman algorithm gives the tree that minimizes the length of the compressed file. See [4, 3] for a detailed description.

Example 1. Consider a text with vocabulary *A, B, C, D, E* where the corresponding frequencies are 0.25, 0.25, 0.20, 0.15, 0.15. A possible Huffman tree, given by the Huffman algorithm, is shown in Figure 1. Observe that *A* is coded with 01, *B* with 10, *C* with 11, *D* with 000 and *E* with 001.
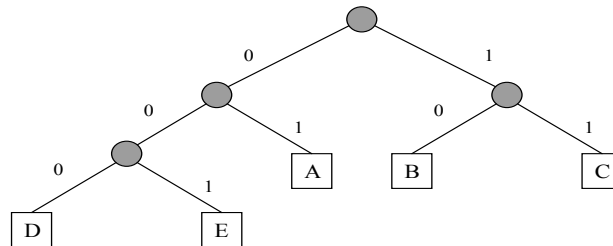
□

Fig. 1. Huffman tree

2.2 Byte-Oriented Huffman Coding

The basic method proposed by Huffman is mostly used as a binary code, that is, each word in the original text is coded as a sequence of bits. Moura et al. [12] modify the code assignment such that a sequence of whole bytes is associated with each word in the text.

Experimental results have shown that, on natural language, there is no significant degradation in the compression ratio by using bytes instead of bits. In addition, decompression and searching are faster with byte-oriented Huffman code because no bit manipulations are necessary.

In [12] two codes following this approach are presented. In that article, they call *Plain Huffman Code* the one we have already described, that is, a word-based byte-oriented Huffman code.

The second code proposed is called Tagged Huffman Code. This is just like the previous one differing only in that the first bit of each byte is reserved to flag whether or not the byte is the first byte of a 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.

Tagged Huffman Code has a price in terms of compression performance: we store full bytes but use only 7 bits for coding. Hence the compressed file grows approximately by 8%–10%.

Example 2. We show the differences among the codes generated by the Plain Huffman Code and Tagged Huffman Code. In our example we assume that the text vocabulary has 16 words, with uniform distribution in Table 1 and with exponential distribution ($p_i = 1/2^i$) in Table 2.

For the sake of simplicity, from this example on, we will consider that our "bytes" are formed by only two bits. Hence, Tagged Huffman Code uses one bit for the flag and one for the code (this makes it look worse than it is). We underline the flag bits. □

Word	Probab.	Plain Huffman	Tagged Huffman
A	1/16	00 00	10 00 00 00
B	1/16	00 01	10 00 00 01
C	1/16	00 10	10 00 01 00
D	1/16	00 11	10 00 01 01
E	1/16	01 00	10 01 00 00
F	1/16	01 01	10 01 00 01
G	1/16	01 10	10 01 01 00
H	1/16	01 11	10 01 01 01
I	1/16	10 00	11 00 00 00
J	1/16	10 01	11 00 00 01
K	1/16	10 10	11 00 01 00
L	1/16	10 11	11 00 01 01
M	1/16	11 00	11 01 00 00
N	1/16	11 01	11 01 00 01
O	1/16	11 10	11 01 01 00
P	1/16	11 11	11 01 01 01

Table 1. Codes for a uniform distribution.

The addition of a tag bit in the Tagged Huffman Code permits direct searching on the compressed text with any string matching algorithm, by simply compressing the pattern and then resorting to classical string matching.

On Plain Huffman this does not work, as the pattern could occur in the text and yet not correspond to our codeword. The problem is that the concatenation of parts of two codewords may form the codeword of another vocabulary word.

This cannot happen in the Tagged Huffman Code due to the use of one bit in each byte to determine if the byte is the first byte of a codeword or not.

Word	Probab.	Plain Huffman	Tagged Huffman
A	1/2	00	11
B	1/4	01	10 01
C	1/8	10	10 00 01
D	1/16	11 00	10 00 00 01
E	1/32	11 01	10 00 00 00 01
F	1/64	11 10	10 00 00 00 00 01
G	1/128	11 11 00	10 00 00 00 00 00 01
H	1/256	11 11 01	10 00 00 00 00 00 00 01
I	1/512	11 11 10	10 00 00 00 00 00 00 00 01
J	1/1024	11 11 11 00	10 00 00 00 00 00 00 00 00 01
K	1/2048	11 11 11 01	10 00 00 00 00 00 00 00 00 00 01
L	1/4096	11 11 11 10	10 00 00 00 00 00 00 00 00 00 00 01
M	1/8192	11 11 11 11 00	10 00 00 00 00 00 00 00 00 00 00 00 01
N	1/16384	11 11 11 11 01	10 00 00 00 00 00 00 00 00 00 00 00 00 01
O	1/32768	11 11 11 11 10	10 00 00 00 00 00 00 00 00 00 00 00 00 00 01
P	1/32768	11 11 11 11 11	10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01

Table 2. Codes for an exponential distribution.

For this reason, searching with Plain Huffman requires inspecting all the bytes of the compressed text from the beginning, while Boyer-Moore type searching (that is, skipping bytes) is possible over Tagged Huffman Code.

Example 3. Let us suppose that we have to compress a text with a vocabulary formed by the words A, B, C, D and assume that the Huffman algorithm assigns the following codewords to the original words:

A	00
B	01
C	10
D	11 00

Let us consider the following portion of a compressed text using the code shown above, for the sequence $ABAD$:

$$\ldots 00\ 01\ 00\ 11\ 00 \ldots$$

Finally, let us suppose that we search for word A. If we resort to plain pattern matching, we find two occurrences in the text. However, the second does not really represent an occurrence of A in the text, but it is part of D. The program should have a postprocessing phase where each potential occurrence is verified, which ruins the simplicity and performance of the algorithm. □

The algorithm to search for a single word under Tagged Huffman Code starts by finding the word in the vocabulary to obtain the codeword that

represents it in the compressed text. Then the obtained codeword is searched for in the compressed text using any classical string matching algorithm with no modifications. They call this technique *direct searching* [12, 17].

Today's IR systems require also flexibility in the search patterns. There is a range of complex patterns that are interesting in IR systems, including regular expressions and "approximate" searching (also known as "search allowing errors"). See [12, 17] for more details.

3   A New Compression Scheme: End-Tagged Dense Codes

We start with a seemingly dull change to Tagged Huffman Code. Instead of using the flag bit to signal the *beginning* of a codeword, we use it to signal the *end* of a codeword. That is, the bit will be 1 for the last byte of each codeword.

This change has surprising consequences. Now the flag bit is enough to ensure that the code is a prefix code, no matter what we do with the other 7 bits. To see this, notice that, given two codewords X and Y, where $|X| < |Y|$, X cannot be a prefix of Y because the last byte of X has its flag bit in 1, while the $|X|$-th byte of Y has its flag bit in 0.

At this point, there is no need at all to use Huffman coding over the remaining 7 bits. We can just use *all* the possible combinations of 7 bits in all the bytes, as long as we reserve the flag bit to signal the end of the codeword.

Once we are not bound to use a Huffman code, we have the problem of finding the optimal code assignment, that is, the one minimizing the length of the output. It is still true that we want to assign shorter codewords to more frequent words. Indeed, the optimal assignment is obtained with the following procedure.

1. The words in the vocabulary are ordered by their frequency, more frequent first.
2. Codewords from $\underline{1}0000000$ to $\underline{1}1111111$ are assigned sequentially to the first 128 words of the vocabulary, using the 2^7 possibilities.
3. Words at positions $128 + 1$ to $128 + 128^2$ are encoded using two bytes, by exploiting the 2^{14} combinations from $\underline{0}0000000:\underline{1}0000000$ to $\underline{0}1111111:\underline{1}1111111$.
4. Words at positions $128 + 128^2 + 1$ to $128 + 128^2 + 128^3$ are encoded using three bytes, by exploiting the 2^{21} combinations from $\underline{0}0000000:\underline{0}0000000:\underline{1}0000000$ to $\underline{0}1111111:\underline{0}1111111:\underline{1}1111111$. And so on.

The assignment is done in a completely sequential fashion, that is, the 130-th word is encoded as $\underline{0}0000000:\underline{1}0000001$, the 131-th as $\underline{0}0000000:\underline{1}0000010$, and so on, just as if we had a 14-bit number. As it can be seen, the computation of codes is extremely simple: It is only necessary to order the vocabulary words by frequency and then sequentially assign the codewords. Hence the coding phase will be faster because obtaining the codes is simpler.
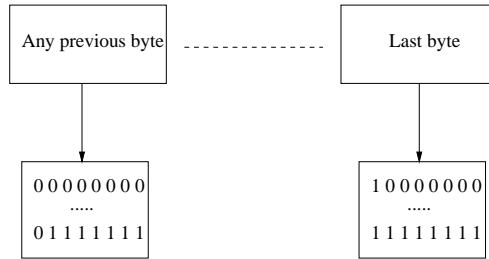
Fig. 2. End-Tagged Dense Codewords

In fact, we do not even need to physically store the results of these computations: With a few operations we can obtain on the fly, given a word rank i, its ℓ-byte codeword, in $O(\ell) = O(\log i)$ time.

What is perhaps less obvious is 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* with frequencies 0.27, 0.26, 0.25 and 0.23, respectively, the code will be the same as if their frequencies were 0.9, 0.09, 0.009 and 0.001.

Hence, we do not 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. Therefore, the vocabulary will be smaller than in the case of the Huffman code, where either the frequencies or the codewords or the tree must be stored with the vocabulary.

In order to obtain the codewords of a compressed text, the decoder can run a simple computation to obtain, from the codeword, the rank of the word, and then obtain the word from the vocabulary sorted by frequency. An ℓ-byte code n can be decoded in $O(\ell) = O(\log n)$ time.

Table 3 shows the codewords obtained by the End-Tagged Dense Code for the examples in Tables 1 and 2 (remember that we are using bytes of two bits). Note that, independently of the distribution, exponential or uniform, the codification is the same.

It is interesting to point out how our codes look like on a Huffman tree. Basically, our codes can be regarded as a tree of arity 128 (like that of Tagged Huffman), but in this case we can use also the *internal* nodes of the tree, while Tagged Huffman is restricted to use only the leaves. This reason shows immediately that the best tree for us is the most balanced one (that is, a dense code), as any tree is legal and the balanced one minimizes the weighted path length.

4   Analytical Results

We try to analyze the compression performance of our new scheme. Let us assume a word distribution $\{p_i\}_{i=1...N}$, where p_i is the probability of the i-th most frequent word and N is the vocabulary size. Let us assume that we use symbols

Word	Rank	Codeword
A	1	10
B	2	11
C	3	00 10
D	4	00 11
E	5	01 10
F	6	01 11
G	7	00 00 10
H	8	00 00 11
I	9	00 01 10
J	10	00 01 11
K	11	01 00 10
L	12	01 00 11
M	13	01 01 10
N	14	01 01 11
O	15	00 00 00 10
P	16	00 00 00 11

Table 3. Example of End-Tagged Dense Code

of b bits to represent the codewords, so that each codeword is a sequence of b-bit symbols. In practice we use bytes, so $b = 8$.

It is well known [3] that Plain Huffman coding produces an average symbol length which is at most one extra symbol over the zero-order entropy. That is, if we call

$$E_b = \sum_{i=1}^{N} p_i \log_{2^b}(1/p_i) = \frac{1}{b} \sum_{i=1}^{N} p_i \log_2(1/p_i)$$

the zero-order entropy in base b of the text, then the average number of symbols to code a word using Plain Huffman is

$$E_b \leq H_b \leq E_b + 1$$

Tagged Huffman code is also easy to analyze. It is a Huffman code over $b-1$ bits, but using b bits per symbol, hence

$$E_{b-1} \leq T_b \leq E_{b-1} + 1$$

Let us now consider our new method, with average number of symbols per word D_b. It is clear that $H_b \leq D_b \leq T_b$, because ours is a prefix code and Huffman is the best prefix code, and because we use all the $b-1$ remaining bit combinations and Tagged Huffman does not. We try now to obtain a more precise comparison. Let us call $B = 2^{b-1}$. Since B^i different words will be coded using i symbols, let us define

$$s_i = \sum_{j=1}^{i} B^j = \frac{B}{B-1}(B^i - 1)$$

(where $s_0 = 0$) the number of words that can be coded with up to i symbols. Let us also call

$$f_i = \sum_{j=s_{i-1}+1}^{s_i} p_j$$

the overall probability of words coded with i symbols.

Then, the average length of a codeword under our new method is

$$D_b = \sum_{i=1}^{S} i f_i$$

where $S = \log_B \left(\frac{B-1}{B} N + 1 \right)$.

The most interesting particular case is a distribution typical of natural language texts. It is well known [3] that, in natural language texts, the vocabulary distribution closely follows a generalized Zipf's law [14], that is, $p_i = A/i^\theta$ and $N = \infty$, for suitable constants A and θ. In practice θ is between 1.4 and 1.8 and depends on the text [1, 2], while

$$A = \frac{1}{\sum_{i \geq 1} 1/i^\theta} = \frac{1}{\zeta(\theta)}$$

makes sure that the distribution adds up 1^1. Under this distribution the entropy is

$$E_b = \frac{1}{b} \sum_{i \geq 1} p_i \log_2 \frac{1}{p_i} = \frac{A\theta}{b} \sum_{i \geq 1} \left(\frac{\log_2 i}{i^\theta} - \log_2 A \right) = \frac{-\theta\zeta'(\theta)/\zeta(\theta) + \ln \zeta(\theta)}{b \ln 2}$$

On the other hand, we have

$$D_b = A \sum_{i \geq 1} i \sum_{j=s_{i-1}+1}^{s_i} 1/j^\theta = 1 + A \sum_{i \geq 1} i \sum_{j=s_i+1}^{s_{i+1}} 1/j^\theta = 1 + A \sum_{i \geq 1} \sum_{j \geq s_i+1} 1/j^\theta$$

At this point we resort to integration to get lower and upper bounds. Since $1/j^\theta$ decreases with j, we have that the above summation is upper bounded as follows

$$D_b \leq 1 + A \sum_{i \geq 1} \int_{s_i}^{\infty} 1/x^\theta dx = 1 + \frac{A(B-1)^{\theta-1}}{(\theta-1)B^{\theta-1}} \sum_{i \geq 1} \frac{1}{(B^i - 1)^{\theta-1}}$$

$$\leq 1 + \frac{A(B-1)^{\theta-1}}{(\theta-1)B^{\theta-1}} \frac{B^{1-\theta}}{1 - B^{1-\theta}} (1 - 1/B)^{1-\theta} = 1 + \frac{1}{(\theta-1)\zeta(\theta)(B^{\theta-1} - 1)}$$

A lower bound can be obtained similarly, as follows

$$D_b \geq 1 + A \sum_{i \geq 1} \int_{s_i+1}^{\infty} 1/x^\theta dx = 1 + \frac{A(B-1)^{\theta-1}}{\theta-1} \sum_{i \geq 1} \frac{1}{(B^{i+1} - 1)^{\theta-1}}$$

$$\geq 1 + \frac{A(B-1)^{\theta-1}}{(\theta-1)} \frac{B^{2(1-\theta)}}{1 - B^{1-\theta}} = 1 + \frac{(1 - 1/B)^{\theta-1}}{(\theta-1)\zeta(\theta)(B^{\theta-1} - 1)}$$

---

[1] We are using the Zeta function $\zeta(x) = \sum_{i>0} 1/i^x$. We will also use $\zeta'(x) = \partial \zeta(x)/\partial x$.

This gives us the length of the code with a precision factor of $(1 - 1/B)^{\theta - 1}$, which in our case ($B = 128$, $\theta = 1.4$ to 1.8) is around 0.5%. This shows that the new code is also simpler to analyze than Huffman, as the existing bounds for Huffman are much looser.

In fact, the lower bound E_b for the performance of Huffman is quite useless for our case: For $b = 8$ bits, $E_b < 1$ for $\theta > 1.3$, where 1 symbol is an obvious lower bound. The same happens when using E_{b-1} to bound Tagged Huffman. We have considered tighter estimates [5, 11] but none was useful for this case.

In order to compare our results with Plain Huffman and Tagged Huffman, we resort to our own analysis. If we take $b = 9$ bits, then $B = 256$ and we obtain the length of a dense code where all the 8 bits are used in the optimal form. This is necessarily better than Huffman (on 8 bits), as not all the 8-bit combinations are legal for Huffman. On the other hand, consider $b = 7$. In this case our new code is a 7-bit prefix code, necessarily inferior to Tagged Huffman (over 8 bits). Hence we have the following inequalities

$$D_{b+1} \quad \leq \quad H_b \quad \leq \quad D_b \quad \leq \quad T_b \quad \leq \quad D_{b-1}$$

These results give us usable bounds to compare our performance. Incidentally, the analysis of our new code turns out to give new upper and lower bounds for the redundancy of d-ary Huffman codes. Figure 3 illustrates our analytical estimates as a function of θ. Our lower and upper bounds are rather close. Plain Huffman must lie between its lower bound and our upper bound. Tagged Huffman must lie between our lower bound and its upper bound.

5   Experimental Results

We show some experimental results now. We have used some large text collections from TREC-4 (Congressional Record 1993, Financial Times 1991 and 1992); and some smaller from the *Calgary Corpus* (Book1, Book2, Paper1 to Paper6, Bib). We have compressed them using Plain Huffman Code, Tagged Huffman Code, and our End-Tagged Dense Code. We have excluded the size of the vocabulary in the results (this size is almost equal in the three cases, although a bit smaller in our case). Separators (that is, maximal strings between two consecutive words) were treated as words as well. Since words and separators strictly alternate, we used separate dictionaries for words and separators, as one can compress and decompress using one after the other.

Table 4 shows the results. It can be seen that, in all cases, our End-Tagged Dense Codes are superior to Tagged Huffman Codes. In particular, on the large natural language collections, we are 5%–6% better than Tagged Huffman and just less than 2% over the optimal Plain Huffman. The results change a bit on the smaller collections of the Calgary Corpus, but essentially the results are the same. Note that the separators are so biased (θ close to 2.0) that they permit an almost optimal coding (1 byte per separator), while words have a smaller θ (1.4–1.6), which makes their code longer (1.5–1.7 bytes per word). Observe that the θ values for the smaller files of the Calgary corpus strongly deviate from

Compression bounds
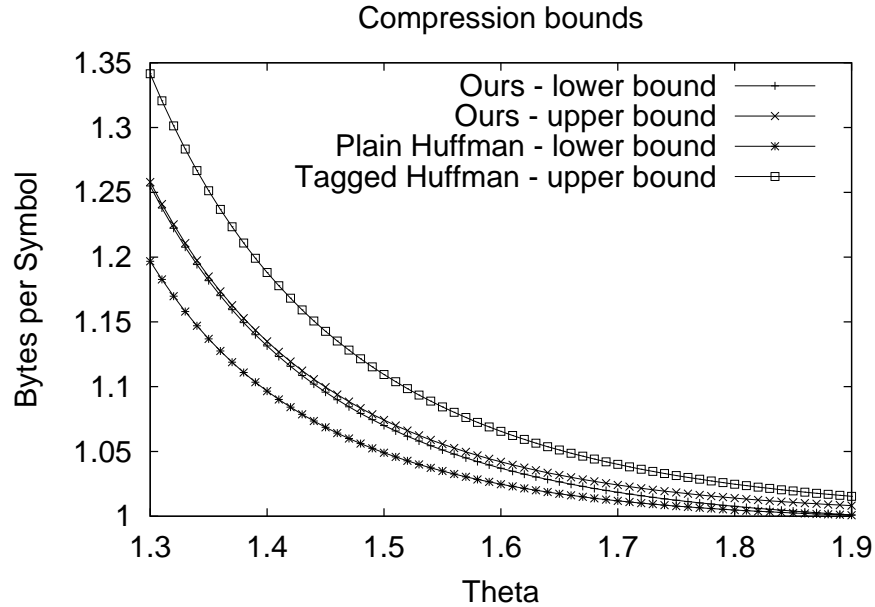
Fig. 3. Analytical bounds on the average code length for byte-oriented Plain Huffman, Tagged Huffman, and our new method. We assume a Zipf distribution with parameter θ (which is the x axis).

these lines and from previous results on other large collections [1], which shows that this kind of analysis applies well to large collections only.

6 Conclusions

We have presented a new compression code useful for text databases. The code inherits from previous work, where byte-oriented word-based Huffman codes were shown to be an excellent choice. To permit fast searching over that code, Tagged Huffman codes were introduced, which in exchange produced an output about 7%–8% larger.

In this paper we have introduced End-Tagged Dense Codes, which is a prefix code retaining all the searchability properties of Tagged Huffman code while improving it on several aspects: (i) codes are shorter: 5%–6% shorter than Tagged Huffman and just less than 2% over Huffman; (ii) coding is much simpler and faster; (iii) the vocabulary representation is simpler and shorter.

We have shown analytically and experimentally the advantages of End-Tagged Dense Codes in terms of output size. An Information Retrieval system based on this new technique should also benefit from the other advantages. For example, (iii) means that we just need to store the vocabulary sorted by frequency, without any additional information, which has an important impact

Corpus	Tot. Words Tot. Sep.	Voc. Words Voc. Sep.	Plain Huffman	End-Tagged Dense	Tagged Huffman	Theta
Congressional Record	7,765,434	112,850	1.543411	1.586550	1.709030	1.623903
1993	7,765,481	1,320	1.000506	1.001422	1.001595	2.319811
Financial Times	2,426,418	74,351	1.539468	1.581893	1.702489	1.438503
1991	2,426,433	1,245	1.000843	1.001705	1.001870	2,003911
Financial Times	28,882,888	279,330	1.552060	1.591722	1.721922	1.627602
1992	28,883,064	5,564	1.001234	1.002280	1.002726	1.775175
Calgary: Book1	31,722	5,565	1.381281	1.460280	1.511727	0.969072
	31,723	317	1.001954	1.007313	1.007439	1.531933
Calgary: Book2	105,962	7,907	1.410222	1.491638	1.564929	1.271044
	105,963	1,114	1.014467	1.027028	1.028444	1.521758
Calgary: Paper1	9,157	1,791	1.300426	1.415420	1.435514	1.004143
	9,158	316	1.006661	1.028281	1.028936	1.426831
Calgary: Paper2	14,266	2,468	1.326090	1.427309	1.453175	1.035512
	14,267	184	1.000000	1.003925	1.003995	1.632045
Calgary: Paper3	7,364	2,087	1.381586	1.483568	1.502308	0.835870
	7,365	125	0.999999	0.999999	0.999999	1.667132
Calgary: Paper4	2,219	751	1,248422	1,385933	1,392696	0.809930
	2,218	75	0.999999	0.999999	0.999999	1.582207
Calgary: Paper5	2,207	613	1.166289	1.299048	1.304486	0.914670
	2,208	190	0.999999	1.028080	1.028533	1.134055
Calgary: Paper6	7,246	1,163	1.213497	1.341844	1.355921	1.098575
	7,247	373	1.016283	1.044708	1.045361	1.263650
Calgary: Bib	20,519	3,667	1.295287	1.365953	1.393391	0.922653
	20,520	56	1.000000	1.000000	1.000000	2.735627
Calgary: News	62,794	9,912	1.511116	1.581616	1.701739	1.025764
	62,795	2,060	1.045242	1.062107	1.065690	1.107732

Table 4. Compression results on different text collections. The compression is expressed in terms of bytes per symbol.

when coding short collections; (*ii*) means that we do not have to build Huffman code, but can just encode and decode on the fly with a program of a few lines.

As a side effect, our analysis has given new upper and lower bounds on the average code length when using d-ary Huffman coding. These bounds are of different nature from those we are aware of, and they could be better on some distribution. This was the case on natural language distributions.

For the final version we will improve the experimental results and their agreement with the analytical predictions. There are different valid ways to compute the θ value for a distribution (we used plain least squares on the model $y = c/x^{\theta}$), and we would like to find one where the θ value is useful to predict the compression obtained according to the analysis. Right now, a smaller θ does not guarantee better compression, as it would if the real distribution fitted perfectly a Zipf distribution. This does not alter the fact that our End-Tagged Dense codes are strictly better than Tagged Huffman codes and that in practice the

improvement is of 5%–6%, but it would be interesting to be able to predict the improvement just from a suitable θ value for the collection.

References

1. M. D. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In R. Baeza-Yates, editor, *Proc. 4th South American Workshop on String Processing (WSP'97)*, pages 2–20. Carleton University Press, 1997.
2. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
3. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.
4. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, September 1952.
5. D. Manstetten. Tight bounds on the redundancy of Huffman codes. *IEEE Trans. on Information Theory*, 38(1):144–151, January 1992.
6. A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.
7. A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. In *Proc. Data Compression Conference*, pages 170–179, 1996.
8. G. Navarro, E. Silva de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.
9. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
10. G. Navarro and J. Tarhio. Boyer-moore string matching over ziv-lempel compressed text. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'2000)*, LNCS 1848, pages 166–180, 2000.
11. R. De Prisco and A. De Santis. On lower bounds for the redundancy of optimal codes. *Designs, Codes and Cryptography*, 15(1):29–45, 1998.
12. E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, April 2000.
13. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, second edition, 1999.
14. G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.
15. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
16. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.
17. N. Ziviani, E. Silva de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.