(S,C)-Dense Coding: An Optimized Compression Code for Natural Language Text Databases *

Nieves R. Brisaboa[1], Antonio Fariña[1], Gonzalo Navarro[2] and María F. Esteller[1]

[1] Database Lab., Univ. da Coruña, Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain. {brisaboa,fari}@udc.es, mfesteller@yahoo.es
[2] Dept. of Computer Science, Univ. de Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl

Abstract. This work presents (s, c)-Dense Code, a new method for compressing natural language texts. This technique is a generalization of a previous compression technique called End-Tagged Dense Code that obtains better compression ratio as well as a simpler and faster encoding than Tagged Huffman. At the same time, (s, c)-Dense Code is a prefix code that maintains the most interesting features of Tagged Huffman Code with respect to direct search on the compressed text. (s, c)-Dense Coding retains all the efficiency and simplicity of Tagged Huffman, and improves its compression ratios.
We formally describe the (s, c)-Dense Code and show how to compute the parameters s and c that optimize the compression for a specific corpus. Our empirical results show that (s, c)-Dense Code improves End-Tagged Dense Code and Tagged Huffman Code, and reaches only 0.5% overhead over plain Huffman Code.

1 Introduction

Text compression techniques are based on exploiting redundancies in the text to represent it using less space [2]. The amount of text collections has grown in the last 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 and the associated costs decrease, the size of text collections increases also faster. 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 at the expense of for some extra CPU time.

Another advantage of text compression techniques is that all of them allow (and improve) the use of *block addressing indexes*. These indexes are smaller than standard inverted indexes because their entries point to blocks instead of exact word positions. Of course the price to pay is sequential text scanning of the

* This work is partially supported by CICYT Grant (#TIC2002-04413-C04-04), CYTED VII.19 RIBIDI Project, and (for the third author) Fondecyt Grant 1-020831.

pointed blocks. However, if the text is compressed with a technique that allows direct search of words in the compressed text, then not only the index and text size are reduced, but also the search inside candidate text blocks is much faster. Notice that using those two techniques together, as in [8], the index is used just as a device to filter out some blocks that do not contain the word we are looking for. This index schema was first proposed in Glimpse [13], a widely known system that uses a block addressing index. On the other hand compression techniques can be used as well to compress the inverted indexes themselves, as suggested in [8] or [10].

For these reasons, 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 of documents will be less efficient, due to the need of decompression before the search.

Classic compression techniques, like 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 [7] 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], they presented 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 [6]) is to take the words as the symbols that compose the text (and therefore the symbols that should be compressed). Since in Information Retrieval (IR) words are the atoms of the search, these compression schemes are particularly suitable for IR.

In [3] it is shown that, although plain Huffman Code is the prefix code that gives the shortest possible output when a source symbol is always substituted by the same code, Tagged Huffman Code largely underutilizes the representation. In that paper it is shown that, by signaling the last byte instead of the first one, the rest of the bits can be used in all their combinations and the code is still a prefix code. The resulting code, called End-Tagged Dense Code, becomes much closer to the compression obtained by Plain Huffman Code. This code not only retains the ability of being searchable with any string matching algorithm, but it is also extremely simple to build (it is not based on Huffman at all) and permits a more compact representation of the vocabulary. Thus, the advantages over Tagged Huffman Code are (*i*) better compression ratios, (*ii*)

same searching possibilities, (*iii*) simpler and faster coding and (*iv*) simpler and smaller vocabulary representation.

In this paper we present (s, c)-Dense Coding, a generalization of the End-Tagged Dense Code [3] that improves its compression ratio by adapting the number of terminal and non terminal symbols to the distribution of frequencies of the words in the corpus to be compressed. As a result, (s, c)-Dense Coding compresses strictly better than End-Tagged Dense Code and Tagged Huffman Code, reaching only a 0.5% overhead over Plain Huffman Code. At the same time, (s, c)-Dense Codes retain all the simplicity and direct search capabilities of End-Tagged Dense Codes and Tagged Huffman Codes. We present an efficient algorithm to build (s, c)-Dense Codes, which is so fast that it can make this an interesting alternative to Plain Huffman Codes because of speed and simplicity of construction.

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.

Plain Huffman Code [3] 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

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

2.1 Word-Based Huffman Compression

The traditional implementations of the Huffman code are character based, i.e., they use the characters as the symbols of the alphabet. In [5] they use 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 more compressible when regarded as a sequence of words rather than characters.

In [12,17], a compression scheme that uses this strategy combined with a Huffman code is presented. 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 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. In [12] they modified the code assignment such that a sequence of bytes instead of bits 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.

2.2 Tagged Huffman Codes

In [12] two codes following this approach are presented. In that paper, they call *Plain Huffman Code* to 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 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 11%.

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

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

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

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.

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 on how Tagged Huffman Code permits these types of search.

2.3 End-Tagged Dense Codes

The End-Tagged Dense Code [3] starts with a seemingly dull change to Tagged Huffman Code. Instead of using the flag bit to signal the *beginning* of a codeword, the flag bit is used to signal the *end* of a codeword. That is, the flag bit is *0* for the first bit of any byte of a codeword except for the last one, which has a *1* in its first bit.

This change has surprising consequences. Now the flag bit is enough to ensure that the code is a prefix code regardless of the contents of the other 7 bits. To see this, consider two codewords X and Y, being X shorter than Y ($|X| < |Y|$). 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. It is possible to use *all* the possible combinations of 7 bits in all the bytes, as long as the flag bit is used to mark the last byte of the codeword.

We are not restricted to use symbols of 8 bits to form the codewords. It is possible to use symbols of b bits. The End-Tagged Dense Code is defined as follows:

Definition 1 Given source symbols with decreasing probabilities $\{p_i\}_{0 \le i < n}$ *the corresponding codeword using the End-Tagged Dense Code is formed by a sequence of symbols of b bits, all of them representing base-*(2^{b-1}) *digits (that is, from 0 to* $2^{b-1} - 1$*), except the last one which has a value between* 2^{b-1} *and* $2^b - 1$*, and the assignment is done in a completely sequential fashion.*

That is, using symbols of 8 bits, the 130-th word is encoded as 00000000:10000001, the 131-th as 00000000:10000010, 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 than using Huffman because obtaining the codes is simpler.

In fact, it is not necessary 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, then the code will be the same as if their frequencies were 0.9, 0.09, 0.009 and 0.001.

Hence, 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. Therefore, the vocabulary will be slightly smaller than in the case of the Huffman code, where some information about the shape of the tree must be stored (even when a canonical Huffman tree is used).

In order to obtain the codewords, 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. A code i of ℓ bytes can be decoded in $O(\ell) = O(\log i)$ time.

An interesting property of this code is that it can be used as a bound for the compression that can be obtained with a Huffman code. It is clear that the End-Tagged Dense Code uses all the possible combinations of all bits, except the first one, that is used as a flag as in the Tagged Huffman Code. Therefore, calling D_b the code length of an End-Tagged Dense Code that uses symbols of b bits we have

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

In [3] a more precise comparison on these three codes is presented. There, it is shown how the End-Tagged Dense Code provides lower and upper bounds to the compression that can be obtained by Huffman with texts in natural language where the Zipf's Law is assumed [14].

3 (s,c)- Dense Codes

As we have shown, End-Tagged Dense Code uses 2^{b-1} digits, from 0 to $2^{b-1}-1$, for the bytes at the beginning of a codeword, and it uses the other 2^{b-1} digits, from 2^{b-1} to $2^b - 1$, for the last byte of the codeword. But the question that arises now is whether that proportion between the number of *non terminal* and *terminal* digits is the optimal one; that is, for a given corpus with a specific distribution of frequencies of words, it might be that a different number of non terminal digits (continuers) and terminal digits (stoppers) could compress better than just using 2^{b-1}. This idea has been previously pointed out in [9]. We define (s, c)- stop-cont codes as follows.

Definition 2 Given source symbols with probabilities $\{p_i\}_{0 \leq i < n}$ *an* (s, c) *stop-cont code (where* c *and* s *are integers larger than zero) assigns to each source symbol* i *a unique target code formed by a base-*c *digit sequence terminated by 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 $c+s-1$, i.e., it is a base-s number that is distinguished from previous digits by adding c. Digits between 0 and $c-1$ are called "continuers" and those between c and $c+s-1$ are called "stoppers" . The next property clearly follows.

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

Proof. If one code were a prefix of the other, since the shorter code must have a final digit of value at least c, then the longer code must have an intermediate digit which is not in base c. This is a contradiction. □

Among all the possible (s, c) stop-cont codes for a given probability distribution, the *dense code* is one that minimizes the average symbol length. This is because a dense code uses all the possible combinations of bits in each byte. That is, codes can be assigned sequentially to the ranked symbols.

Definition 3 Given source symbols with decreasing *probabilities* $\{p_i\}_{0 \le i < n}$*, the corresponding* (s, c)*-Dense Code (*(s, c)*-DC) is an* (s, c) *stop-cont code where the codewords are assigned as follows: Let* k *be the number of bytes in each codeword, which is always* ≥ 1*, then* k *will be such that*

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

Thus, the code corresponding to source symbol i is formed by $k-1$ base-c digits and a final digit. If $k = 1$ then the code is simply the stopper $c+i$. Otherwise the code is formed by the number $\lfloor x/s \rfloor$ written in base c, followed by $c+(x \ mod \ s)$, where $x = i - \frac{sc^{k-1}-s}{c-1}$.

Example 1. The codes assigned to symbols $i \ \epsilon \ 0 \ldots 15$ by a (2,3)-DC are as follows: ⟨3⟩, ⟨4⟩, ⟨0,3⟩, ⟨0,4⟩, ⟨1,3⟩, ⟨1,4⟩, ⟨2,3⟩, ⟨2,4⟩, ⟨0,0,3⟩, ⟨0,0,4⟩, ⟨0,1,3⟩, ⟨0,1,4⟩, ⟨0,2,3⟩, ⟨0,2,4⟩, ⟨1,0,3⟩ and ⟨1,0,4⟩.

Note that the code does not depend on the exact symbol probabilities, but just 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 Example 1, together with the symbol they encode, if any. Then it is clear that (i) any unused code 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 symbol with a shorter code is swapped with a more probable symbol with a longer code then the average code length decreases, and hence sorting the symbols by decreasing frequency is optimal. □

Since sc^{k-1} different codewords can be coded using k digits, let us call

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

(where $W_0^s = 0$) the number of source symbols that can be coded with up to k digits. Let us also call

$$f_k^s = \sum_{j=W_{k-1}^s+1}^{W_k^s} p_j$$

the sum of probabilities of source symbols coded with k digits by an (s, c)-DC.

Then, the average codeword length for the (s, c)-DC, $LD_{(s,c)}$, is

$$\begin{aligned} LD_{(s,c)} &= \sum_{k=1}^{K^s} k f_k^s = \sum_{k=1}^{K^s} k \sum_{j=W_{k-1}^s+1}^{W_k^s} p_j \\ &= 1 + \sum_{k=1}^{K^s-1} k \sum_{j=W_k^s+1}^{W_{k+1}^s} p_j = 1 + \sum_{k=1}^{K^s-1} \sum_{j=W_k^s+1}^{W_k^s} p_j \end{aligned}$$

where $K^x = \lceil \log_{(2^b-x)} \left(1 + \frac{n(2^b-x-1)}{x}\right) \rceil$, and n is the number of symbols in the vocabulary.

It is clear from Definition 3 that the End-Tagged Dense Code [3] is a (2^{b-1},2^{b-1})-DC and therefore *(s,c)*-DC can be seen as a generalization of the End-Tagged Dense Code where s and c are adjusted to optimize the compression for the distribution of frequencies of the vocabulary.

In [3] it is proved that (2^{b-1},2^{b-1})-DC is more efficient than Tagged Huffman. This is because Tagged Huffman is a (2^{b-1},2^{b-1}) *(non dense) stop-cont* code, while the End-Tagged Dense Code is a (2^{b-1},2^{b-1})-Dense Code.

Example 2. Table 1 shows the codewords assigned to a small set of words ordered by their frequency when using Plain Huffman (P.H.), *(6,2)*-DC, End-Tagged Dense Code (ETDC) which is a *(4,4)*-DC, and Tagged Huffman (TH). Symbols of three bits are used for simplicity (b=*3*). The last four columns present the products of the number of bytes by the frequency for each word, and its addition, the average codeword length, is shown in the last row.

It is easy to see that, for this example, Plain Huffman and the (6,2)-Dense Code are better than the (4,4)-Dense Code (ETDC) and therefore they are also better than Tagged Huffman. Notice that (6,2)-Dense Code is clearly better than (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 an optimal compressed text having, in this example, the same result than Plain Huffman.

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.

Word	Freq	P.H.	(6,2)-DC	ETDC	T.H.	Freq × bytes			
						P.H.	(6,2)-DC	ETDC	T.H.
A	0.2	[000]	[010]	[100]	[100]	0.2	0.2	0.2	0.2
B	0.2	[001]	[011]	[101]	[101]	0.2	0.2	0.2	0.2
C	0.15	[010]	[100]	[110]	[110]	0.15	0.15	0.15	0.3
D	0.15	[011]	[101]	[111]	[111][000]	0.15	0.15	0.15	0.3
E	0.14	[100]	[110]	[000][100]	[111][001]	0.14	0.14	0.28	0.28
F	0.09	[101]	[111]	[000][101]	[111][010]	0.09	0.09	0.18	0.18
G	0.04	[110]	[000][010]	[000][110]	[111][011][000]	0.04	0.08	0.08	0.12
H	0.02	[111][000]	[000][011]	[000][111]	[111][011][001]	0.04	0.04	0.04	0.05
I	0.005	[111][001]	[000][100]	[001][100]	[111][011][010]	0.01	0.01	0.01	0.015
J	0.005	[111][010]	[000][101]	[001][101]	[111][011][011]	0.01	0.01	0.01	0.015
					total compressed size	**1.03**	**1.07**	**1.30**	**1.67**

Table 1. Comparative among compression methods

4 Optimal s and c Values

Before giving the algorithm to compute the optimal s and c values, $s+c = 2^b$ and $1 \leq s \leq 2^b - 1$, we need to show that the size of the compressed text decreases when we increase s until reaching the *unique* optimal s value. After that optimal value, increments of s will produce a loss in the compression ratio. This is shown in Figure 1. Of course the value of c depends on the value of s because $c = 2^b - s$ always holds.

Although we have a formal proof of the uniqueness of the minimum, it is so technically involved that it could be an article by itself. We considered such a long proof inadequate for this conference version. So we have decided to include only an intuitive explanation in this paper, which turns out to be considerably simpler than the actual proof with all the details.

Intuitively, it is easy to see that when s is very small the number of high frequency words encoded with very few bytes (that is, one or two bytes) is also very small (s words are encoded with just one byte and $s \cdot c$ with two bytes) but in this case c is large and therefore words with low frequency will be encoded with few bytes ($s \cdot c^2$ words will be encoded with 3 bytes, $s \cdot c^3$ with 4 bytes and so on, but if c is so large, probably 3 bytes will be enough to encode the last word of the ranked vocabulary).

It is clear that, as s grows, highest frequency words will be encoded with less bytes, so we improve the compression of high frequency words. But at the same time, as s grows, lowest frequency words will need more bytes to be encoded, so we loss compression in those words.

As consequence, if we try all the possible values of s starting at $s = 1$, we will see (as in Figure 1) that, in the beginning, compression improves a lot because each increment of s produce that words with high frequency become encoded by a codeword that is one byte shorter.

When s becomes larger, for each increment of s the number of words encoded with less bytes is smaller in proportion and has lower frequency. Therefore, with

each increment of s, we gain less and less compression in the highest frequency words. At the same time, we lose more and more compression in the lowest frequency words, because with each increment of s they will need more bytes to be encoded. At some 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 decreases. That point gives us the optimal s value. It is easy to see in Figure1 that, around of the optimal value, the compression is relatively insensitive to the exact value of s. This fact causes the smooth bottom of the curve.

Our algorithm takes advantage of this property. It is not necessary to check all the values of s because we know the shape of the distribution of compression ratios as a function of s. Our algorithm looks only for the direction of change the in value of s, moving towards the area where compression ratio improves.

5 Algorithm to Obtain the Optimal s and c Values

In this section, an algorithm to find the best s and consequently c for a given corpus and a given b is shown. A needed precondition is to have a list with the accumulated frequencies for all the words in the corpus, arranged in decreasing order of frequency.

The basic algorithm is presented below. It is a binary search algorithm that initially computes the size of the compressed text for two consecutive values of s: ($\lfloor 2^{b-1} \rfloor - 1$ and $\lfloor 2^{b-1} \rfloor$). As has been intuitively explained there is at most one local minimum, so the algorithm can lead the search to the point that reaches the best compression ratio. In each new iteration the search space is reduced by half and a new computation of the compression is obtained with two central points of the new interval is performed.

The process consists of two parts: The algorithm **findBestS** computes the best s and c values for a given b and a list of accumulated frequencies. This list can be obtained by sorting the words in decreasing order of frequency and computing the accumulated frequency for each position. The *size of the compressed text* (in bytes) is computed for specific s and c values by calling function **ComputeSizeS**. Finally, the s and c values that minimize the length are returned.

```
findBestS (b, freqList)
  //Input: b value (2^b = c + s).
  //Input: A list of accumulated frequencies for words arranged on decreasing order of frequency .
  //Output: The best s and c values
  begin
    Lp := 1;              //Lp and Up the lower and upper
    Up := 2^b − 1;        //points of the interval being checked
    while Lp + 1 < Up do
        M := ⌊(Lp+Up)/2⌋;
        sizePp := computeSizeS(M − 1, 2^b − (M − 1), freqList); //size with M − 1
        sizeM := computeSizeS(M, 2^b − M, freqList);            //size with M
        if sizePp < sizeM then
            Up := M − 1;
```

```
      else Lp := M
      end if
    end while
    if Lp < Up then     //Lp = Up − 1 and M = Lp
      sizeNp := computeSizeS(Up, 2^b − Up, freqList); //size with M + 1
      if sizeM < sizeNp then
        bestS := M;
      else bestS := Up
      end if
    else bestS := Lp     //Lp = Up = M − 1
    end if
    bestC := 2^b − bestS;
    return bestS, bestC;
  end
```

For any given values s and c, next algorithm computes the size of the compressed text.

```
computeSizeS (s, c, freqList)
  //Inputs: s, c, and a list of accumulated frequencies.
  //Output: length of the compressed text when using s,c
  begin
    k := 1; n := number ofwords in 'freqList';
    Right := min(s, n);      total := freqList[Right − 1];
    while Right < n do
        Left := Right;
        Right := Right + sc^k;
        k := k + 1;
        if Right > n then
            Right := n;
        end if
        total := total + k ∗ (freqList[Right − 1] − freqList[Left]) ;
    end while
    return total
  end
```

Notice that computing the size of the compressed text for a specific value of s costs $O(\log_c n)$, except for $c = 1$, in which case it costs $O(n/s) = O(n/2^b)$. Hence the most expensive possible sequence of calls to **computeSizeS** in a binary search is that for values $c = 2^{b-1}$, $c = 2^{b-2}$, $c = 2^{b-3}$, ..., $c = 1$. The total cost of **computeSizeS** over that sequence of c values is

$$\frac{n}{2^b} + \sum_{i=1}^{b-1} \log_{2^{b-i}} n \;=\; \frac{n}{2^b} + \log_2 n \sum_{i=1}^{b-1} \frac{1}{b-i} \;=\; O\left(\frac{n}{2^b} + \log n \log b\right)$$

The other operations of the binary search are constant, and we have also an extra $O(n)$ cost to compute the accumulated frequencies. Hence the overall cost to find s and c is $O(n+\log(n)\log(b))$. Since the maximum b of interest is such that $b = \lceil \log_2 n \rceil$ (as at this point we can code each symbol using a single stopper), the optimization algorithm costs at most $O(n + \log(n)\log\log(n)) = O(n)$, assuming the vocabulary is already sorted. We have succeeded in making the optimization part totally negligible. Huffman algorithm is also linear once the vocabulary is sorted, but the constant is in practice larger because it involves more operations than just adding up frequencies.

6 Empirical Results

We used some large text collections from TREC-2 (AP Newswire 1988 and Ziff Data 1989-1990) and from TREC-4 (Congressional Record 1993, Financial Times 1991, 1992, 1993 and 1994). We also used a Literary Spanish corpus we created. We have compressed them using Plain Huffman, *(s,c)*-Dense Code, End-Tagged Dense Code and Tagged Huffman. We used the spaceless word model [11] to create the vocabulary; that is, if a word was followed by a space, we just encoded the word, otherwise both the word and the separator were encoded.

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 (s, c)-DC and ETDC because only the ranking of words is needed).

Corpus	Num words	vocabulary size	Entropy	Plain	(s,c)-DC	ETDC	Tagged
AP 1988	52,960,212	268,890	1,3032	1,4778	(189,67) 1,4908	1,5166	1,6382
Ziff 1989-90	40,548,114	237,607	1,2732	1,4500	(198,58) 1,4586	1,4909	1,6064
C.R. 1993	9,445,990	117,713	1,2950	1,4755	(195,61) 1,4874	1,5202	1,6381
F.T. 1991	3,059,634	75,687	1,2825	1,4555	(192,64) 1,4674	1,4974	1,6122
F.T. 1992	36,518,075	284,878	1,2946	1,4651	(193,63) 1,4755	1,5042	1,6275
F.T. 1993	41,772,135	291,404	1,2839	1,4474	(195,61) 1,4566	1,4891	1,6131
F.T. 1994	43,039,879	294,990	1,2843	1,4476	(195,61) 1,4569	1,4897	1,6137
Spanish Texts	18,324,100	313,977	1,3612	1,5486	(182,74) 1,5692	1,5889	1,7164

Table 2. Comparison of the Average Codeword Length

In Table 2 we present the average length of the codewords obtained with each of the compression methods, as well as the zero-order entropy of the text when words are taken as the source symbols. As it can be seen, the average codeword length in Plain Huffman and in (s, c)-Dense Code is less than 1 byte larger than the entropy. The sixth column, also gives the optimal (s, c) values found using algorithm **findBestS**.

Corpus	Original Size	Voc. Words	θ	Entropy	P.H.	(s,c)-DC	ETDC	T.H.
AP 1988	250,994,525	241,315	1.852045	27.49	31.18	(189,67) 31.46	32.00	34.57
Ziff 1989-90	185,417,980	221,443	1.744346	27.84	31.71	(198,58) 31.90	32.60	35.13
C.R. 1993	51,085,545	114,174	1.634076	23.94	27.28	(195,61) 27.50	28.11	30.29
F.T. 1991	14,749,355	75,597	1.449878	26.60	30.19	(193,63) 30.44	31.06	33.44
F.T. 1992	175,449,248	284,904	1.630996	26.94	30.49	(193,63) 30.71	31.31	33.88
F.T. 1993	197,586,334	291,322	1.647456	27.14	30.60	(195,61) 30.79	31.48	34.10
F.T. 1994	203,783,923	295,023	1.649428	27.12	30.57	(195,61) 30.77	31.46	34.08
Spanish Texts	105,125,124	313,977	1.480535	23.71	27.00	(182,74) 27.36	27.71	29.93

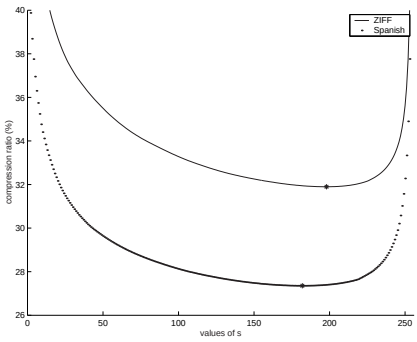
Table 3. Comparison of compression ratios.

Table 3 shows the compression ratio obtained by the aforementioned codes, as well as that corresponding to the entropy. The second column contains the original size of the processed corpus and the following columns indicate the number of words in the vocabulary, the θ parameter of Zipf's Law [14,1], and

the compression ratio for each method. Again, in the column corresponding to (s, c)-DC, the optimal (s, c) values are shown.

As it can be seen, Plain Huffman gets the best compression ratio (as expected since it is the optimal prefix code) and End-Tagged Dense Codes always obtain better results than Tagged Huffman, with an improvement of up to 2.5 points. As expected, (s, c)-DC improves the results reached by $(128, 128)$-DC (ETDC). In fact, (s, c)-DC is superior to $(128, 128)$-DC and it is worse than the optimal Plain Huffman only by less than 0.5 points on average.

In Figure 1 the size of the compressed texts and the compression ratios are shown as a function of the s values, for Ziff and Spanish corpus. As shown in Table 3, the optimal s value for Ziff corpus is 198, while for the Spanish corpus the maximum compression ratio is achieved with $s = 182$. On the right we show sizes and compression ratios when s values close to the optimum are used.



s value	ZIFF Corpus		Spanish Corpus	
	ratio	size	ratio	size
175	31.9891	59,313,463	27.3592	28,761,439
180	31.9550	59,250,251	27.3530	28,754,926
181	31.9489	59,239,066	27.3527	28,754,571
182	31.9432	59,228,446	**27.3526**	**28,754,487**
183	31.9378	59,218,406	27.3529	28,754,757
185	31.9279	59,200,048	27.3543	28,756,250
190	31.9092	59,165,319	27.3633	28,765,740
195	31.8990	59,146,488	27.3814	28,784,711
197	31.8976	59,143,905	27.3915	28,795,340
198	**31.8976**	**59,143,837**	27.3972	28,801,304
199	31.8980	59,144,550	27.4033	28,807,734
200	31.8987	59,146,012	27.4098	28,814,637

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

7 Conclusions

We have presented (s, c)-Dense Codes, a new method for compressing natural language texts. This method is a generalization of the End-Tagged Dense Code and improves its compression ratio by adapting its (s, c) parameters to the corpus to be compressed.

We have given an algorithm that computes the optimal s, c values for a given corpus, that is, the pair that maximizes the compression ratio. Instead of sequentially computing the resulting size for each s and c value, and then choosing the best one, our algorithm uses the fact that there is a unique minimum in the size of the compressed text as a function of the s and c values, to speed up the process. In fact, our algorithm has an $O(\log n \times \log \log n)$ cost. We have presented an intuitive description of this nontrivial property of the behavior of minima as a function of s.

We have presented some empirical results comparing our method against other codes with similar features. Our new code is always better than End-

Tagged Dense Code and Tagged Huffman Code, reaching only 0.5% excess from the optimal Huffman Code. It is also faster and simpler to build.

It seems that there should be some relationship between the θ of the Zipf's model (more or less biased distribution of frequencies) and the optimal s and c values. It can be shown that $s = 2^b - 2^{b/\theta}$. However, our empirical data do not confirm that. The reason is that the Zipf model is a rough approximation, possibly useless for this case. On the other hand, we found that s is always in the interval $[182, 198]$, and this information could be used to speed up, a little bit, our algorithm. In order to generalize this result we would need a model (working in practice) which, from the frequency distribution, gave us a range where the optimum s must lie.

References

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman, May 1999.
2. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, 1990.
3. N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In *25th European Conference on IR Research, ECIR 2003*, LNCS 2633, pages 468–481, 2003.
4. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, 1952.
5. A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.
6. A. Moffat and A. Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.
7. 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.
8. Gonzalo Navarro, Edleno Silva de Moura, Marden Neubert, Nivio Ziviani, and Ricardo Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.
9. J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, LNCS 2373, pages 42–52, 2002.
10. F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. 25th Annual International ACM SIGIR conference on Research and development in information retrieval*, pages 222–229, 2002.
11. E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR-98)*, pages 298–306, 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, 2000.
13. Manber U. and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. of the Winter 1994 USENIX Technical Conference*, pages 23–32, 1994.

14. G.K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.
15. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
16. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.
17. N. Ziviani, E. Silva de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *Computer*, 33(11):37–44, 2000.