Simple, Fast, and Efficient Natural Language Adaptive 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. One of the most successful natural language compression methods is word-based Huffman. However, such a two-pass semi-static compressor is not well suited to many interesting real-time transmission scenarios. A one-pass adaptive variant of Huffman exists, but it is character-oriented and rather complex. In this paper we implement word-based adaptive Huffman compression, showing that it obtains very competitive compression ratios. Then, we show how End-Tagged Dense Code, an alternative to word-based Huffman, can be turned into a faster and much simpler adaptive compression method which obtains almost the same compression ratios.

1 Introduction

Transmission of compressed data is usually composed of four processes: *compression*, *transmission*, *reception*, and *decompression*. The first two are carried out by a *sender* process and the last two by a *receiver*. This abstracts from communication over a network, but also from writing a compressed file to disk so as to load and decompress it later. In some scenarios, especially the latter, compression and transmission usually complete before reception and decompression start.

There are several interesting *real-time* transmission scenarios, however, where those processes should take place concurrently. That is, the sender should be able to start the transmission of compressed data without preprocessing the whole text, and simultaneously the receiver should start reception and decompress the text as it arrives. Real-time transmission is usually of interest when communicating over a network. This kind of compression can be applied, for example, to interactive services such as remote login or talk/chat protocols, where small messages are exchanged during the whole communication time. It might also be relevant to transmission of Web pages, so that the exchange of

* This word is partially supported by CYTED VII.19 RIBIDI Project. It is also funded in part (for the Spanish group) by MCyT (PGE and FEDER) grant(TIC2003-06593) and (for the third author) by Millennium Nucleus Center for Web Research, Grant (P01-029-F), Mideplan, Chile.

(relatively small) pages between a server and a client along the time enables adaptive compression by installing a browser plug-in to handle decompression. This might be also interesting for wireless communication with hand-held devices with little bandwidth and processing power.

Real-time transmission is handled with so-called *dynamic* or *adaptive* compression techniques. These perform a single pass over the text (so they are also called *one-pass*) and begin compression and transmission as they read the data. Currently, the most widely used adaptive compression techniques belong to the Ziv-Lempel family [1]. When applied to natural language text, however, the compression ratios achieved by Ziv-Lempel are not that good (around 40%).

Statistical *two-pass* techniques, on the other hand, use a *semi-static* model. A first pass over the text to compress gathers global statistical information, which is used to compress the text in a second pass. The computed model is transmitted prior to the compressed data, so that the receiver can use it for decompression. Classic Huffman code [11] is a well-known two-pass method. Its compression ratio is rather poor for natural language texts (around 60%). In recent years, however, new Huffman-based compression techniques for natural language have appeared, based on the idea of taking the words, not the characters, as the source symbols to be compressed [13]. Since in natural language texts the frequency distribution of words is much more biased than that of characters, the gain in compression is enormous, achieving compression ratios around 25%-30%. Additionally, since in Information Retrieval (IR) words are the atoms searched for, these compression schemes are well suited to IR tasks. Word-based Huffman variants focused on fast retrieval are presented in [7], where a byte- rather than bit-oriented coding alphabet speeds up decompression and search.

Two-pass codes, unfortunately, are not suitable for real-time transmission. Hence, developing an adaptive compression technique with good compression ratios for natural language texts is a relevant problem. In [8, 9] a dynamic Huffman compression method was presented. This method was later improved in [12, 14]. In this case, the model is not previously computed nor transmitted, but rather computed and updated on the fly both by sender and receiver.

However, those methods are character- rather than word-oriented, and thus their compression ratios on natural language are poor. Extending those algorithms to build a dynamic word-based Huffman method and evaluating its compression efficiency and processing cost is the first contribution of this paper. We show that the compression ratios achieved are in most cases just 0.06% over those of the semi-static version. The algorithm is also rather efficient: It compresses 4 megabytes per second in our machine. On the other hand, it is rather complex to implement.

Recently, a new word-based byte-oriented method called End-Tagged Dense Code (ETDC) was presented in [3]. ETDC is not based on Huffman at all. It is simpler and faster to build than Huffman codes, and its compression ratio is only 2%-4% over the corresponding word-based byte-oriented Huffman code. For IR purposes, ETDC is especially interesting because it permits direct text

searching, much as the Tagged Huffman variants developed in [7]. However, ETDC compresses better than those fast-searchable Huffman variants.

The second contribution of this paper is to show another advantage of ETDC compared to Huffman codes. We show that an adaptive version of ETDC is much simpler to program and 22%-26% faster than word-oriented dynamic Huffman codes. Moreover, its compression ratios are only 0.06% over those of semi-static ETDC, and 2%-4% over semi-static Huffman code. From a theoretical viewpoint, dynamic Huffman complexity is proportional to the number of target symbols output, while dynamic ETDC complexity is proportional to the number of source symbol processed. The latter is never larger than the former, and the difference increases as more compression is obtained.

As a sanity check, we also present empirical results comparing our dynamic word-based codes against two well-known compression techniques such as *gzip* (fast compression and decompression, but poor compression) and *bzip2* (good compression ratio, but slower). These results show that our two techniques provide a well balanced trade-off between compression ratio and speed.

2 Word-Based Semi-Static Codes

Since in this paper we focus on word-based natural language text compression, we speak indistinctly of *source symbols* and *words*, and sometimes call *vocabulary* the set of source symbols.

2.1 Word-Based Huffman Codes

The idea of Huffman coding [11] is to compress the text by assigning shorter codes to more frequent symbols. Huffman algorithm obtains an optimal (shortest total length) *prefix code* for a given text. A code is a *prefix 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.

The word-based Huffman byte oriented codes proposed in [7] obtain compression ratios on natural language close to 30% by coding with bytes instead of bits (in comparison to the bit oriented approach that achieves ratios close to 25%). In exchange, decompression and searching are much faster with byte-oriented Huffman code because no bit manipulations are necessary. This word-based byte-oriented Huffman code will be called *Plain Huffman* code in this paper.

Another code proposed in [7] is *Tagged Huffman* code. This is like Plain Huffman, except that the first bit of each byte is reserved to flag whether the byte is the first of its 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. The tag bit permits direct searching on the compressed text by simply compressing the pattern and then running any classical string matching algorithm. On Plain

Huffman this does not work, as the pattern could occur in the text not aligned to any codeword [7].

While searching Plain Huffman compressed text requires inspecting all its bytes from the beginning, Boyer-Moore type searching (that is, skipping bytes) [2] is possible over Tagged Huffman code. On the other hand, Tagged Huffman code pays a price in terms of compression performance of approximately 11%, as it stores full bytes but uses only 7 bits for coding.

2.2 End-Tagged Dense Codes

End-Tagged Dense code (ETDC) [3] is obtained by a seemingly dull change to Tagged Huffman code. Instead of using a flag bit to signal the *beginning* of a codeword, the *end* of a codeword is signaled. That is, the highest bit of any codeword byte is 0 except for the last byte, where it is 1. By this change there is no need at all to use Huffman coding in order to maintain a prefix code.

In general, ETDC can be defined over 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 Given source symbols $\{s_1, \ldots, s_n\}$*, End-Tagged Dense Code assigns number* $i-1$ *to the* i*-th most frequent symbol. This number is represented in base* 2^{b-1}*, as a sequence of digits, from most to least significant. Each such digit is represented using* b *bits. The exception is the least significant digit* d_0*, where we represent* $2^{b-1} + d_0$ *instead of just* d_0*.*

That is, the first word 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 coded 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 $\underline{0}1111111$:$\underline{1}1111111$, 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 sort the source symbols by decreasing frequency and then sequentially assign the codewords. The coding phase is faster than using Huffman because obtaining the codes is simpler. Empirical results comparing ETDC against Plain and Tagged Huffman can be found in [3].

Note that the code depends on the rank of the words, not on their actual frequency. As a result, it is not even necessary to transmit the code of each word, but just the sorted vocabulary, as the model to the receiver. Hence, End-Tagged Dense Codes are simpler, faster, and compress better than Tagged Huffman codes. Since the last bytes of codewords are distinguished, they also permit direct search of the compressed text for the compressed pattern, using any search algorithm.

On-the-fly Coding and Decoding. We finally observe that, for compression and decompression, we do not really have to start by sequentially assigning the codes to the sorted words. An on-the-fly encoding is also possible.

```
Sender ( )
(1)  Vocabulary ← {C_new-Symbol};
(2)  Initialize CodeBook;
(3)  for i ∈ 1...n do
(4)    read s_i from the text;
(5)    if s_i ∉ Vocabulary then
(6)      send C_new-Symbol;
(7)      send s_i in plain form;
(8)      Vocabulary ← Vocabulary ∪ {s_i};
(9)      f(s_i) ← 1;
(10)   else
(11)     send CodeBook(s_i);
(12)     f(s_i) ← f(s_i) + 1;
(13)   Update CodeBook;
```

```
Receiver ( )
(1)  Vocabulary ← {C_new-Symbol};
(2)  Initialize CodeBook;
(3)  for i ∈ 1...n do
(4)    receive C_i;
(5)    if C_i = C_new-Symbol then
(6)      receive s_i in plain form;
(7)      Vocabulary ← Vocabulary ∪ {s_i};
(8)      f(s_i) ← 1;
(9)    else
(10)     s_i ← CodeBook^-1(C_i);
(11)     f(s_i) ← f(s_i) + 1;
(12)   output s_i;
(13)   Update CodeBook;
```

Fig. 1. Sender and receiver processes in statistical dynamic text compression.

Given a word ranked i in the sorted vocabulary, the encoder can run a simple *encode* function to compute the codeword $C_i = encode(i)$. It is a matter of expressing $i - 1$ in base 2^{b-1} (which requires just bit shifts and masking) and outputting the sequence of digits. Function *encode* takes just $O(l)$ time, where $l = O(\log(i)/b)$ is the length in digits of codeword C_i.

At decompression time, given codeword C_i of l digits and the sorted vocabulary, it is also possible to compute, in $O(l)$ time, function $i = decode(C_i)$, essentially by interpreting C_i as a base 2^{b-1} number and finally adding 1. Then, we retrieve the i-th word in the sorted vocabulary.

3 Statistical Dynamic Codes

Statistical dynamic compression techniques are one-pass. Statistics are collected as the text is read, and consequently, the model is updated as compression progresses. They do not transmit the model, as the receiver can figure out the model by itself from the received codes.

In particular, *zero-order* compressors model the text using only the information on source symbol frequencies, that is, $f(s_i)$ is the number of times source symbol s_i appears in the text (read up to now). In the discussion that follows we focus on zero-order compressors.

In order to maintain the model up to date, dynamic techniques need a data structure to keep the vocabulary of all symbols s_i and their frequencies $f(s_i)$ up to now. This data structure is used by the encoding/decoding scheme, and is continuously updated during compression/decompression. After each change in the vocabulary or frequencies, the codewords assigned to *all* source symbols may have to be recomputed due to the frequency changes. This recomputation must be done both by the sender and the receiver.

Figure 1 depicts the sender and receiver processes, highlighting the symmetry of the scheme. *CodeBook* stands for the model, used to assign codes to source symbols or vice versa.

3.1 Dynamic Huffman Codes

In [8, 9] an adaptive character-oriented Huffman coding algorithm was presented. It was later improved in [12], being named *FGK* algorithm. *FGK* is the basis of the UNIX *compact* command.

FGK maintains a Huffman tree for the source text already read. The tree is adapted each time a symbol is read to keep it optimal. It is maintained both by the sender, to determine the code corresponding to a given source symbol, and by the receiver, to do the opposite. Thus, the Huffman tree acts as the *CodeBook* of Figure 1. Consequently, it is initialized with a unique special node called *zeroNode* (corresponding to *new-Symbol*), and it is updated every time a new source symbol is inserted in the vocabulary or a frequency increases. The codeword for a source symbol corresponds to the path from the tree root to the leaf corresponding to that symbol. Any leaf insertion or frequency change may require reorganizing the tree to restore its optimality.

The main challenge of Dynamic Huffman is how to reorganize the Huffman tree efficiently upon leaf insertions and frequency increments. This is a complex and potentially time-consuming process that must be carried out both by the sender and the receiver.

The main achievement of *FGK* is to ensure that the tree can be updated by doing only a constant amount of work per node in the path from the affected leaf to the tree root. Calling $l(s_i)$ the path length from the leaf of source symbol s_i to the root, and $f(s_i)$ its frequency, then the overall complexity of algorithm *FGK* is $\sum f(s_i)l(s_i)$, which is exactly the length of the compressed text, measured in number of target symbols.

3.2 Word-Based Dynamic Huffman Codes

We implemented a word-based version of algorithm *FGK*. This is by itself a contribution because no existing adaptive technique obtains similar compression ratio on natural language. As the number of text words is much larger than the number of characters, several challenges arised to manage such a large vocabulary. The original *FGK* algorithm pays little attention to these issues because of its underlying assumption that the source alphabet is not very large.

However, the most important difference between our word-based version and the original *FGK* is that we chose the code to be byte rather than bit-oriented. Although this necessarily implies some loss in compression ratio, it gives a decisive advantage in efficiency. Recall that the algorithm complexity corresponds to the number of target symbols in the compressed text. A bit-oriented approach requires time proportional to the number of bits in the compressed text, while ours requires time proportional to the number of bytes. Hence byte-coding is almost 8 times faster.

Being byte-oriented implies that each internal node can have up to 256 children in the resulting Huffman tree, instead of 2 as in a binary tree. This required extending *FGK* algorithm in several aspects.

4 Dynamic End-Tagged Dense Code

In this section we show how ETDC can be made adaptive. Considering again the general scheme of Figure 1, the main issue is how to maintain the *CodeBook* up to date upon insertions of new source symbols and frequency increments. In the case of ETDC, the *CodeBook* is essentially the array of source symbols sorted by frequencies. If we are able to maintain such array upon insertions and frequency changes, then we are able to code any source symbol or decode any target symbol by using the on-the-fly *encode* and *decode* procedures explained at the end of Section 2.2.

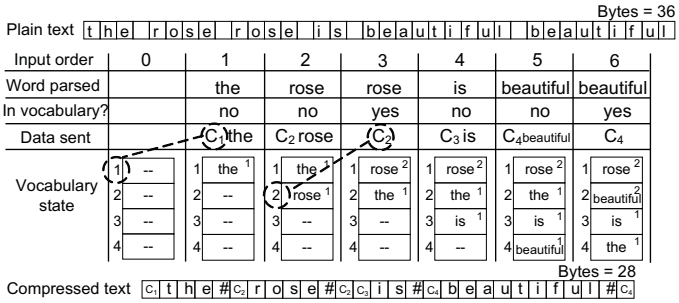

Fig. 2. Transmission of message `"the rose rose is beautiful beautiful"`

Figure 2 shows how the compressor operates. At first (step 0), no words have been read so *new-Symbol* is the only word in the vocabulary (it is implicitly placed at position 1). In step 1, a new symbol `"the"` is read. Since it is not in the vocabulary, C_1 (the codeword of *new-Symbol*) is sent, followed by `"the"` in plain form (bytes `'t'`, `'h'`, `'e'` and some terminator `'#'`). Next, `"the"` is added to the vocabulary (array) with frequency 1, at position 1. Implicitly, *new-Symbol* has been displaced to array position 2. Step 2 shows the transmission of `"rose"`, which is not yet in the vocabulary. In step 3, `"rose"` is read again. As it was in the vocabulary at array position 2, only codeword C_2 is sent. Now, `"rose"` becomes more frequent than `"the"`, so it moves upward in the array. Note that a hypothetical new occurrence of `"rose"` would be transmitted as C_1, while it was sent as C_2 in step 1. In steps 4 and 5, two more new words, `"is"` and `"beautiful"`, are transmitted and added to the vocabulary. Finally, in step 6, `"beautiful"` is read again, and it becomes more frequent than `"is"` and `"the"`. Therefore, it moves upward in the vocabulary by means of an exchange with `"the"`.

The main challenge is how to efficiently maintain the sorted array. In the sequel we show how we obtain a complexity equal to the number of source symbols transmitted. This is always lower than *FGK* complexity, because at

least one target symbol must be transmitted for each source symbol, and usually several more if some compression is going to be achieved. Essentially, we must be able to identify *groups* of words with the same frequency in the array, and be able of fast promoting of a word to the next group when its frequency increases.

The data structures used by the sender and their functionality are shown in Figure 3. The hash table of words keeps in *word* the source word characters, in *posInVoc* the position of the word in the vocabulary array, and in *freq* its frequency. In the vocabulary array (*posInHT*) the words are not explicitly represented, but a pointer to *word* is stored. Finally, arrays *top* and *last* tell, for each possible frequency, the vocabulary array positions of the first and last word with that frequency. It always holds $top[f-1] = last[f]+1$ (so actually only one array is maintained). If no words of frequency f exist, then $last[f] = top[f]-1$.

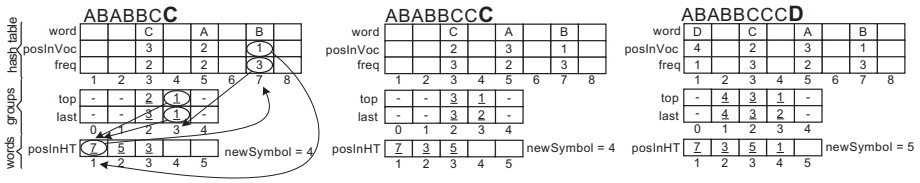

Fig. 3. Transmission of words: ABABBCC, ABABBCCC and ABABBCCCD.

When the sender reads word s_i, it uses the hash function to obtain its position p in the hash table, so that $word[p] = s_i$. After reading $f = freq[p]$, it increments $freq[p]$. The index of s_i in the vocabulary array is also obtained as $i = posInVoc[p]$ (so it will send code C_i). Now, word s_i must be promoted to its next group. For this sake, it finds the head of its group $j = top[f]$ and the corresponding word position $h = posInHT[j]$, so as to swap words i and j in the vocabulary array. The swapping requires exchanging $posInHT[j]$ with $posInHT[i]$, setting $posInVoc[p] = j$ and setting $posInVoc[h] = i$. Once the swapping is done, we promote j to the next group by setting $last[f+1] = j$ and $top[f] = j+1$.

If s_i turns out to be a new word, we set $word[p] = s_i$, $freq[p] = 1$, and $posInVoc[p] = n$, where n is the number of source symbols known prior to reading s_i (and considering *new-Symbol*). Then exactly the above procedure is followed with $f = 0$ and $i = n$. Also, n is incremented.

The receiver works very similarly, except that it starts from i and then it obtains $p = posInHT[i]$. Figure 4 shows the pseudocode.

Implementing dynamic ETDC is simpler than building dynamic word-based Huffman. In fact, our implementation of the Huffman tree update takes about 120 C source code lines, while the update procedure takes only about 20 lines in dynamic ETDC.

Sender (s_i)
(1) $p \leftarrow hash(s_i)$;
(2) **if** $word[p] = nil$ **then** // new word
(3) $word[p] \leftarrow s_i$;
(4) $freq[p] \leftarrow 0$;
(5) $posInVoc[p] \leftarrow n$;
(6) $posInHT[n] \leftarrow p$;
(7) $n \leftarrow n + 1$;
(8) $f \leftarrow freq[p]$;
(9) $freq[p] \leftarrow freq[p] + 1$;
(10) $i \leftarrow posInVoc[p]$;
(11) $j \leftarrow top[f]$;
(12) $h \leftarrow posInHT[j]$;
(13) $posInHT[i] \leftrightarrow posInHT[j]$;
(14) $posInVoc[p] \leftarrow j$;
(15) $posInVoc[h] \leftarrow i$;
(16) $last[f + 1] \leftarrow j$;
(17) $top[f] \leftarrow j + 1$;

Receiver (i)
(1) $p \leftarrow posInHT[i]$;
(2) **if** $word[p] = nil$ **then** // new word
(3) $word[p] \leftarrow s_i$;
(4) $freq[p] \leftarrow 0$;
(5) $posInVoc[p] \leftarrow n$;
(6) $posInHT[n] \leftarrow p$;
(7) $n \leftarrow n + 1$;
(8) $f \leftarrow freq[p]$;
(9) $freq[p] \leftarrow freq[p] + 1$;
(10) $i \leftarrow posInVoc[p]$;
(11) $j \leftarrow top[f]$;
(12) $h \leftarrow posInHT[j]$;
(13) $posInHT[i] \leftrightarrow posInHT[j]$;
(14) $posInVoc[p] \leftarrow j$;
(15) $posInVoc[h] \leftarrow i$;
(16) $last[f + 1] \leftarrow j$;
(17) $top[f] \leftarrow j + 1$;

Fig. 4. Sender and receiver processes to update *CodeBook* in ETDC.

CORPUS	TEXT SIZE	Plain Huffman			End-Tagged Dense Code			diff_{ETDC}
		2-pass	dynamic	Increase	2-pass	dynamic	Increase	
	bytes	ratio %	ratio %	diff_{PH}	ratio %	ratio %	diff_{ETDC}	$-\ \text{diff}_{PH}$
CALGARY	2,131,045	46.238	46.546	0.308	47.397	47.730	0.332	0.024
FT91	14,749,355	34.628	34.739	0.111	35.521	35.638	0.116	0.005
CR	51,085,545	31.057	31.102	0.046	31.941	31.985	0.045	-0.001
FT92	175,449,235	32.000	32.024	0.024	32.815	32.838	0.023	-0.001
ZIFF	185,220,215	32.876	32.895	0.019	33.770	33.787	0.017	-0.002
FT93	197,586,294	31.983	32.005	0.022	32.866	32.887	0.021	-0.001
FT94	203,783,923	31.937	31.959	0.022	32.825	32.845	0.020	-0.002
AP	250,714,271	32.272	32.294	0.021	33.087	33.106	0.018	-0.003
ALL_FT	591,568,807	31.696	31.710	0.014	32.527	32.537	0.011	-0.003
ALL	1,080,719,883	32.830	32.849	0.019	33.656	33.664	0.008	-0.011

Table 1. Compression ratios of dynamic versus semi-static techniques.

5 Empirical Results

We tested the different compressors over several texts. As representative of short texts, we used the whole Calgary corpus. We also 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 to 1994). Finally, two larger collections, ALL_FT and ALL, were used. ALL_FT aggregates all texts from Financial Times collection. ALL collection is composed by Calgary corpus and all texts from TREC-2 and TREC-4.

A dual Intel®Pentium®-III 800 Mhz system, with 768 MB SDRAM-100Mhz was used in our tests. It ran Debian GNU/Linux (kernel version 2.2.19). The compiler used was gcc version 3.3.3 20040429 and `-O9` compiler optimizations were used. Time results measure CPU user-time. The spaceless word model [6] was used to model the separators.

Table 1 compares the compression ratios of two-pass versus one-pass techniques. Columns labeled **diff** measure the increase, in percentual points, in the compression ratio of the dynamic codes compared against their semi-static version. The last column shows those differences between Plain Huffman and ETDC.

To understand the increase of size of dynamic versus semi-static codes, two issues have to be considered: (i) each new word s_i parsed during dynamic

CORPUS	TEXT SIZE	n	Dyn PH		Dyn ETDC		Increase	Decrease
	bytes		time (sec)	ratio%	time (sec)	ratio %	size %	time %
CALGARY	2,131,045	30,995	0.520	46.546	0.384	47.730	2.543	22.892
FT91	14,749,355	75,681	3.428	34.739	2.488	35.638	2.588	22.685
CR	51,085,545	117,713	11.450	31.102	8.418	31.985	2.839	22.629
FT92	175,449,235	284,892	41.330	32.024	31.440	32.838	2.542	26.404
ZIFF	185,220,215	237,622	44.628	32.895	33.394	33.787	2.710	22.559
FT93	197,586,294	291,427	47.118	32.005	36.306	32.887	2.755	20.840
FT94	203,783,923	295,018	48.260	31.959	36.718	32.845	2.774	22.006
AP	250,714,271	269,141	60.702	32.294	47.048	33.106	2.514	22.796
ALL_FT	591,568,807	577,352	143.050	31.710	111.068	32.537	2.609	23.796
ALL	1,080,719,883	886,190	268.983	32.849	213.068	33.664	2.481	25.927

Table 2. Comparison between dynamic ETDC and dynamic PH.

compression is represented in the compressed text (or sent to the receiver) as a pair $\langle C_{new\text{-}Symbol}, s_i \rangle$, while in two-pass compression only the word s_i needs to be stored/transmitted in the vocabulary; (*ii*) on the other hand, some low-frequency words can be encoded with shorter codewords by dynamic techniques, since by the time they appear the vocabulary may still be small.

Compression ratios are around 30-35% for the larger texts. For the smaller ones, compression is poor because the size of the vocabulary is proportionally too large with respect to the compressed text size (as expected from Heaps' law [10]). This means that proportionally too many words are transmitted in plain form.

The increase of size of the compressed texts in ETDC compared to PH is always under 1 percentage point, in the larger texts. On the other hand, the dynamic versions lose very little in compression (maximum 0.02 percentage points, 0.06%) compared to their semi-static versions. This shows that the price paid by dynamism in terms of compression ratio is negligible. Note also that in most cases, and in the larger texts, dynamic ETDC loses even less compression than dynamic Plain Huffman.

Table 2 compares the time performance of our dynamic compressors. The latter two columns measure the increase in compression ratio (in percentage) of ETDC versus Plain Huffman, and the reduction in processing time, in percentage.

As it can be seen, dynamic ETDC loses less than 1 percentage point (3%) of compression ratio compared to dynamic Plain Huffman, in the larger texts. In exchange, it is 22%-26% faster and considerably simpler to implement. Dynamic Plain Huffman compresses 4 megabytes per second, while dynamic ETDC reaches 5.

Tables 3 and 4 compare both dynamic Plain Huffman and dynamic ETDC against *gzip* (Ziv-Lempel family) and *bzip2* (Burrows-Wheeler [5] type technique). Experiments were run setting *gzip* and *bzip2* parameters to "best" (-b) and "fast" (-f) compression.

As expected "bzip2 -b" achieves the best compression ratio. It is about 6 and 7 percentage points better than dynamic PH and dynamic ETDC respectively. However, it is much slower than the other techniques tested in both compression and decompression processes. Using the "fast" *bzip2* option

CORPUS	TEXT SIZE	compression ratio %					
	bytes	Dyn PH	Dyn ETDC	gzip -f	gzip -b	bzip2 -f	bzip2 -b
CALGARY	2,131,045	46.546	47.730	43.530	36.840	32.827	28.924
FT91	14,749,355	34.739	35.638	42.566	36.330	32.305	27.060
CR	51,085,545	31.102	31.985	39.506	33.176	29.507	24.142
FT92	175,449,235	32.024	32.838	42.585	36.377	32.369	27.088
ZIFF	185,220,215	32.895	33.787	39.656	32.975	29.642	25.106
FT93	197,586,294	32.005	32.887	40.230	34.122	30.624	25.322
FT94	203,783,923	31.959	32.845	40.236	34.122	30.535	25.267
AP	250,714,271	32.294	33.106	43.651	37.225	33.260	27.219
ALL_FT	591,568,807	31.710	32.537	40.988	34.845	31.152	25.865
ALL	1,080,719,883	32.849	33.664	41.312	35.001	31.304	25.981

Table 3. Comparison of compression ratio against gzip and bzip2.

seems to be undesirable, since compression ratio gets worse (it becomes closer to dynamic PH) and compression and decompression speeds remain poor.

On the other hand, "gzip -f" is shown to achieve good compression speed, at the expense of compression ratio (about 40%). It is shown that dynamic ETDC is also a fast technique. It is able to beat "gzip -f" in compression speed (except in the ALL corpus). Regarding to compression ratio, dynamic ETDC achieves also best results than "gzip -b" (except in CALGARY and ZIFF corpora). However, *gzip* is clearly the fastest method in decompression.

Hence, dynamic ETDC is either much faster or compresses much better than *gzip*, and it is by far faster than *bzip2*.

CORPUS	compression time (sec)					decompression time (sec)				
	Dyn PH	Dyn ETDC	gzip -f	bzip2 -f	bzip2 -b	Dyn PH	Dyn ETDC	gzip -f	bzip2 -f	bzip2 -b
CALGARY	0,498	0,384	0,360	2,180	2,660	0,330	0,240	0,090	0,775	0,830
FT91	3,218	2,488	2,720	14,380	18,200	2,350	1,545	0,900	4,655	5,890
CR	10,880	8,418	8,875	48,210	65,170	7,745	5,265	3,010	15,910	19,890
FT92	42,720	31,440	34,465	166,310	221,460	30,690	19,415	8,735	57,815	71,050
ZIFF	43,122	33,394	33,550	174,670	233,250	30,440	11,690	9,070	58,790	72,340
FT93	45,864	36,306	36,805	181,720	237,750	32,780	21,935	10,040	62,565	77,860
FT94	47,078	36,718	37,500	185,107	255,220	33,550	22,213	10,845	62,795	80,370
AP	60,940	47,048	50,330	231,785	310,620	43,660	27,233	15,990	81,875	103,010
ALL_FT	145,750	91,068	117,255	558,530	718,250	104,395	66,238	36,295	189,905	235,370
ALL	288,778	213,905	188,310	996,530	1342,430	218,745	126,938	62,485	328,240	432,390

Table 4. Comparison of compression and decompression time against gzip and bzip2.

6 Conclusions

In this paper we have considered the problem of providing adaptive compression for natural language text, with the combined aim of competitive compression ratios and good time performance.

We built an adaptive version of word-based Huffman codes. For this sake, we adapted an existing algorithm so as to handle very large sets of source symbols and byte-oriented output. The latter decision sacrifices some compression ratio in exchange for an 8-fold improvement in time performance. The resulting algorithm obtains compression ratio very similar to its static version (0.06% off) and compresses about 4 megabytes per second on a standard PC.

We also implemented a dynamic version of the End-Tagged Dense Code (ETDC). The resulting adaptive version is much simpler than the Huffman-

based one, and 22%-26% faster, compressing typically 5 megabytes per second. The compressed text is only 0.06% larger than with semi-static ETDC and 3% larger than with Huffman.

As a result, we have obtained adaptive natural language text compressors that obtain 30%-35% compression ratio, and compress more than 4 megabytes per second. Empirical results show their good performance when they are compared against other compressors such as *gzip* and *bzip2*.

Future work involves building an adaptive version of (s, c)-Code [4], an extension to ETDC where the number of byte values that signal the end of a codeword can be adapted to optimize compression, instead of being fixed at 128 as in ETDC. An interesting problem in this case is how to efficiently maintain the optimal (s, c), which now vary as compression progresses.

References

1. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.
2. R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.
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. N.R. Brisaboa, A. Fariña, G. Navarro, and M.F. Esteller. (s,c)-dense coding: An optimized compression code for natural language text databases. In *Proc. 10^{th} International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, pages 122–136, 2003.
5. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.
6. E. 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.
7. E. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.
8. N Faller. An adaptive system for data compression. In *In Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pages 593–597, 1973.
9. R.G Gallager. Variations on a theme by Huffman. *IEEE Trans. on Inf. Theory*, 24(6):668–674, 1978.
10. H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, New York, 1978.
11. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, 1952.
12. D.E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 2(6):163–180, 1985.
13. A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.
14. J.S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM (JACM)*, 34(4):825–845, 1987.