SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2008; **0**:1–23 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

New Adaptive Compressors for Natural Language Text†

N. R. Brisaboa[1], A. Fariña[1,*], G. Navarro[2] and J. R. Parama[1]

[1] *Database Laboratory, Department of Computer Science, University of A Coruña. Campus de Elviña s/n, 15071, A Coruña, Spain.*
[2] *Center for Web Research, Department of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile.*

SUMMARY

Semistatic byte-oriented word-based compression codes have been shown to be an attractive alternative to compress natural language text databases, because of the combination of speed, effectiveness, and direct searchability they offer. In particular, our recently proposed family of *dense* compression codes has been shown to be superior to the more traditional byte-oriented word-based Huffman codes in most aspects. In this paper, we focus on the problem of transmitting texts among peers that do not share the vocabulary. This is the typical scenario for adaptive compression methods. We design adaptive variants of our semistatic dense codes, showing that they are much simpler and faster than dynamic Huffman codes and reach almost the same compression effectiveness. We show that our variants have a very compelling trade-off between compression/decompression speed, compression ratio and search speed compared with most of the state-of-the-art general compressors.

KEY WORDS: Text databases; Natural language text compression; Dynamic compression; Searching compressed text.

1. Introduction

Text compression is of special interest in data transmission. In some scenarios, it is feasible that compression and transmission complete before reception and decompression start. In these cases, statistical *two-pass* techniques, also called *semistatic*, can be used. A first pass over the text gathers global statistical information about the vocabulary (list of source symbols) in order

*Correspondence to: fari@udc.es
†A preliminary partial version on this work appeared in [4].
Contract/grant sponsor: Funded in part (for the Spanish group) by MEC (TIN2006-15071-C03-03), Xunta de Galicia (PGIDIT05-SIN-10502PR) and (for the third author) by Millennium Nucleus Center for Web Research, grant (P04-067-F), Mideplan, Chile.

Copyright © 2008 John Wiley & Sons, Ltd.
Received <Date>
Revised <Date>
Accepted <Date>

to obtain a *model* of the text. The model is used to compute the codeword corresponding to each source symbol and then, in a second pass, each original symbol is substituted by its codeword. Therefore, the model must be stored/transmitted with the compressed text, so the decompressor can know the model to perform the decompression.

In *real-time* transmission, the sender should be able to start the transmission of compressed data without preprocessing the whole text, and simultaneously the receiver should start the reception and decompression of the text as it arrives. 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. Adaptive or dynamic compression methods do not need to transmit the model because the receiver can learn it as it receives the compressed text.

In recent years, statistical semistatic compression techniques especially designed for natural language texts have not only proven extremely effective (with compression ratios around 25%-30%), but also permitted searching the compressed text much faster (up to 8 times) than the original text. The success of these techniques is based in regarding the text to be compressed as a sequence of words instead of characters [2]. In [15], a word based Huffman code, which reaches 25% of compression ratio, was presented. Moura *et. al.* also presented important improvements in two compression codes called *Plain Huffman (PH)* and *Tagged Huffman (TH)* [17]. The byte-oriented Plain Huffman achieves compression ratios close to 30%, as opposed to the 25% that is achieved by using bit-oriented codes [20]. In exchange, decompression is much faster because bit manipulations are not necessary. Tagged Huffman adds a flag bit to mark the limits of each codeword and, therefore, it permits fast direct search of the compressed text, reaching compression ratios around 34%. Due to the flag bit, a pattern can be compressed and directly searched for in the compressed text without decompressing it. This property is also essential to permit local decompression of text passages in order to present them to the final users.

Recently, a family of compression codes called *Dense Codes* has been shown to offer several advantages over Huffman-based compression for natural language [6]. Dense codes are simpler and faster to build than Huffman codes, and they permit the same fast direct searchability of Tagged Huffman, yet with better compression ratios. The simplest variant is *End-Tagged Dense Code (ETDC)*, which is just a variable-length integer representation for the position of the word in a frequency rank. A more sophisticated variant is (s, c)-*Dense Code (SCDC)*, which adapts better to the text distribution. ETDC reaches around 31% compression ratio and SCDC reaches less than 0.3 percentage points over PH compression ratio. Another recent competitive semistatic proposal is the *Restricted Prefix Byte Code* [9], which gets better compression ratio than SCDC but, as PH, does not use a flag bit to mark the limits of each codeword in the compressed text. This absence implies that searchers and decompressors have more difficulties to perform random access and local decompression of text passages due to a problem of synchronization.

Among the adaptive compressors, dynamic arithmetic coding over PPM-like modelling [16] obtains compression ratios around 24%, but it requires significant computational effort by both the sender and the receiver, being quite slow at both ends.

Compression methods based on the Ziv-Lempel family [24, 25] (used in *zip*, *gzip*, *arj*, *winzip*, etc.) obtain reasonable but not spectacular compression ratios on natural language text (around 40%), yet they are very fast at decompression.

Copyright © 2008 John Wiley & Sons, Ltd.

Prepared using speauth.cls

In this paper we introduce two dynamic compressors, called *Dynamic End-Tagged Dense Code (DETDC)* and *Dynamic (s,c)-Dense Code (DSCDC)*, which adapt ETDC and SCDC to real-time transmission. We also implemented a byte-oriented word-based dynamic Huffman compressor, which we call DPH, to have a powerful statistical compressor to compare with our dense dynamic compressors. Details about its implementation can be found in [10].

In this paper, we show experimentally that DETDC and DSCDC offer several advantages over state-of-the-art adaptive compression methods in scenarios where real-time transmission of natural text is needed. Some concrete examples of these scenarios follow:

1. *News agency*: This type of organizations are continuously disseminating news, in real time, to newspapers, TV channels, radio stations, etc. Each piece of news is broadcast to all the registered organizations, and this process is done in real time as news arise.
2. *Digital library*: When a user chooses a literary work, the digital library usually offers the user to download the literary work split in some sort of parts (sections, chapters, pages, etc.). After choosing one of these parts, the server sends the associated text. Those parts can be requested in an arbitrary order.
3. *Chat session* established between two Internet users: Again, the messages are short, and they should be delivered as soon as they are written.
4. *HTTP session* established between a server and a client: The HTML pages are sent by the server when the client requests them.

All those situations can be described as scenarios where a sender sends short messages to a receiver during a certain period of time (session). The individual messages are not long enough to obtain good compression ratios using word-based semistatic compression, as they need to process at least 5-10 Mbytes to compensate for the burden of storing the vocabulary (the model), according to Moura *et. al.* [17] and our own results. The whole session is long enough. This type of transmission must be carried out in real time, therefore it is not feasible to accumulate short messages along time so as to send them together using a semistatic compressor. Therefore, dynamic compression turns out to be the most suitable alternative.

In dynamic compression, the model changes each time a text word is processed. These frequent changes of the model make it difficult to carry out direct searches over text compressed with adaptive methods, as the search pattern looks different in different points of the compressed text. Yet, there are several adaptive compression scenarios where a direct search on the compressed text (without decompressing it) is of interest. For example, for classification or distribution purposes, the receiver may be interested not in uncompressing all the arriving text, but in searching it for some specific words. In ubiquitous computation or mobile databases, servers broadcast information to the devices (PDAs, mobile phones, etc.) in their cell. Probably, these devices are not interested in decompressing all the information they receive. So they would perform a multipattern search on the arriving compressed text seeking some keywords, which denote topics of interest (for example sports, tourism or traffic information). When some of the keywords are found, the device decompresses the information and stores it or points it to specific places in the device.

In all those cases, documents are received and pointed out to users, or stored in specific places, when keywords that denote topics of interest are found in the message. There exist

Copyright © 2008 John Wiley & Sons, Ltd.

Prepared using speauth.cls

some direct search techniques for adaptive compression that, even being slower than searching the uncompressed text, are faster than *uncompressing plus searching* [19]. In this paper we also show how direct search (without decompression) can be done over text compressed with our adaptive dense compressors DETDC and DSCDC. In practice, searches over text compressed with DETDC are faster than searches over text compressed with previous non-dense adaptive techniques. Moreover, our searches are more efficient than searches over uncompressed text when a large number of patterns are searched for, which is usually the case in the applications given above.

The outline of this paper is as follows. In Sections 2 and 3, we describe our two adaptive techniques, DETDC and DSCDC, with sufficient detail to be useful for a practitioner. In Section 4, we briefly comment the advantages and disadvantages of the block-wise versions of the semistatic alternatives. Section 5 presents the experimental results comparing our methods, in terms of compression ratio and compression/decompression speed, against several state-of-the-art compressors. In Section 6, it is shown how to search text compressed with either DETDC or DSCDC without previously decompressing it. We present experimental results comparing those searchers against other search algorithms that work over compressed and uncompressed text. Finally, Section 7 gives our conclusions and future work.

2. Dynamic End-Tagged Dense Codes

2.1. End Tagged Dense Codes

As explained in [17], PH is simply a word-based byte-oriented Huffman code. TH reserves the first bit of each byte 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. While searching PH compressed text requires inspecting all its bytes from the beginning, the tag bit in TH permits a Boyer-Moore-type searching [3] (that is, skipping bytes) by simply compressing the pattern and then running the string matching algorithm. On PH this does not work, as the pattern could occur in the text not aligned to any codeword [17].

ETDC has, as TH, a flag bit, but now this bit signals the *end* of a codeword. That is, the leading bit of a codeword byte is 1 for the *last* byte (not the first) and 0 for the others. The remaining 7 bits of each byte are the responsible for carrying the information. Observe that the flag bit is enough to ensure that the code is a prefix code regardless of the content of the other 7 bits of each byte. Therefore, there is no need at all to use Huffman coding in order to maintain a prefix code. Thus, all the possible combinations of bits can be used to fill the remaining 7 bits of each byte. ETDC obtains better performance than TH in all aspects, whereas it maintains all its good search capabilities.

In fact, the coding scheme used by ETDC had already been used previously to compress integers, such as the document identifiers in inverted indexes [9, 21], receiving different names like *bc* or *variable-byte coding* (*Vbyte*).

Copyright © 2008 John Wiley & Sons, Ltd.
Prepared using speauth.cls

In ETDC, the model is just the vocabulary sorted by frequency, because the codeword assigned to each source word depends only on the rank of such a word in the vocabulary ordered by frequency, and not on its actual frequency.

Definition 1. *Given source symbols with* nonincreasing *probabilities* $\{p_i\}_{0 \leq i < n}$, *the corresponding ETDC codeword for the symbol in position* i *has* k *bytes* ($k \geq 1$), *for the* k *that satisfies:*

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

Thus, the codeword corresponding to source symbol i *is formed by* $k-1$ *digits in base* 2^{b-1}, *and a final base-*2^{b-1} *digit added to* 2^{b-1}. *If* $k=1$ *then the codeword is simply* $i+2^{b-1}$. *Otherwise the codeword is formed by the number* x *written in base* 2^{b-1}, *where* $x = i - \frac{2^{(b-1)k}-2^{b-1}}{2^{b-1}-1}$, *and adding* 2^{b-1} *to the last digit.*

ETDC can be defined over symbols of b bits, although the byte-oriented version ($b=8$) is the most common one. That is, the first word ($i=0$) is encoded as $\langle 128 \rangle$, the second ($i=1$) as $\langle 129 \rangle$, until the 128th as $\langle 255 \rangle$. The 129th word ($i=128$) is encoded as $\langle 0{:}128 \rangle$, the 130th as $\langle 0{:}129 \rangle$ and so on until the $(128^2+128)^{th}$ word $\langle 127{:}255 \rangle$.

The simplicity of the code also allows simple encode and decode procedures, and makes ETDC codification faster than those based in Huffman, since it does not have to deal with a tree. We denote *encode* as the function that obtains the codeword $C_i = encode(i)$ for a word at the i-th position in the ranked vocabulary; *decode* computes the position $i = decode(C_i)$ in the rank, for a codeword C_i. Both functions take just $O(l)$ time, where $l = O(\log(i)/b)$ is the length in digits of codeword C_i, and are efficiently implemented through bit shifts and masking. These algorithms are based on Definition 1.

A complete description of ETDC as well as empirical results comparing ETDC against PH and TH can be found in [6].

2.2. Towards Dynamic End-Tagged Dense Code (DETDC)

The main challenge to make ETDC dynamic is how to maintain the model updated as compression progresses, since this process implies the insertion of new source symbols and frequency increments. In the case of ETDC, the model is essentially the array of source symbols sorted by frequency, therefore this array must be kept ordered upon insertions and frequency changes.

Both sender (compressor) and receiver (decompressor) increase the frequency of a word each time it arrives, and maintain the vocabulary ordered by frequency, carrying out two symmetric processes. Therefore, the sender does not transmit the model, since the receiver can figure it out by itself from the received codewords. The sender only informs the receiver of new source symbols appearing in the text using a special codeword that we denote $C_{zeroNode}$. The sender transmits $C_{zeroNode}$ followed by the source word in ASCII. The receiver inserts it in its vocabulary and sets its frequency to 1. In DETDC, $C_{zeroNode}$ is always the first unused codeword, that is, the codeword that follows that of the last word in the vocabulary. When a

Copyright © 2008 John Wiley & Sons, Ltd.

Plain text: l a n d f a r f a r a w a y l o n g l o n g — Bytes = 27

Input order	0	1	2	3	4	5	6
Word parsed		land	far	far	away	long	long
In vocabulary?		no	no	yes	no	no	yes
Data sent		C_0 land	C_1 far	C_1	C_2 away	C_3 long	C_3
Vocabulary state 0	--	land^1	land^1	far^2	far^2	far^2	far^2
Vocabulary state 1	--	--	far^1	land^1	land^1	land^1	long^2
Vocabulary state 2	--	--	--	--	away^1	away^1	away^1
Vocabulary state 3	--	--	--	--	--	long^1	land^1

Compressed text: C_0 l a n d # C_1 f a r # C_1 C_2 a w a y # C_3 l o n g # C_3 — Bytes = 25

Figure 1. Transmission of `"land far far away long long (ago)"`.

word arrives, and it is already in the vocabulary, the sender transmits its codeword, increases its frequency and reorders the vocabulary if necessary. When the receiver gets a codeword other than $C_{zeroNode}$, it just decodes it to obtain the corresponding vocabulary position, recovers the word and increases its frequency, reordering the vocabulary if necessary.

Figure 1 shows how the compressor operates. At first (step 0), no words have been read, so *zeroNode* is the only word in the vocabulary (it is implicitly placed at position 0). In step 1, a new symbol `"land"` is read. Since it is not in the vocabulary, C_0 (the codeword of *zeroNode*) is sent, followed by `"land"`. Then `"land"` is added to the vocabulary with frequency 1, at position 0. Step 2 shows the transmission of `"far"`, which was not in the vocabulary yet. In step 3, `"far"` is read again. Since it was in the vocabulary at position 1, the codeword C_1 is sent. Now `"far"` becomes more frequent than `"land"`, so it moves upwards in the ordered vocabulary. Note that a hypothetical new occurrence of `"far"` would be transmitted as C_0, although it was sent as C_1 in step 3. In steps 4 and 5, two more new words, `"away"` and `"long"`, are transmitted and added to the vocabulary. Finally, in step 6, `"long"` is read again, and when its frequency is updated, it becomes more frequent than `"away"` and `"land"`. Therefore, it moves upwards in the vocabulary by means of an exchange with `"land"` (which is the first word in the ranked vocabulary with its same frequency).

The main issue is how to efficiently maintain the vocabulary sorted. We show next how to do this with a complexity equal to the number of source symbols transmitted. Essentially, we must be able to identify *blocks* of words with the same frequency in the ordered vocabulary, and to quickly promote a word to the next block when its frequency increases. Promoting a word w_i with frequency f to the next frequency $(f+1)$ block consists of:

- Sliding w_i over all words whose frequency is f. This implies two operations:
  - Locating the first word in the ordered vocabulary whose frequency is f. This word is called top_f.
  - Exchanging w_i with top_f.
- Increasing the frequency of w_i.

Copyright © 2008 John Wiley & Sons, Ltd.
Prepared using speauth.cls

2.3. Data structures for DETDC

The sender maintains a hash table that permits fast searching for a source word s_i. The hash table is also used to obtain the rank i in the vocabulary vector (remember that, to encode a word s_i, using ETDC, only its rank i is needed), as well as its current frequency f_i (which is used to rapidly find the position of word top_{f_i}).

The receiver does not need to maintain a hash table to hold words because finding a word lexicographically is never necessary at decompression. It only needs to use a *word* vector where words are kept sorted by frequency, because the decoding process uses the codeword to directly obtain the rank value i that can be used to index the *word* vector.

Let n be the vocabulary size and F the maximum frequency value for any word in the vocabulary. The data structures used by both the sender and the receiver, as well as their functionality, are given next.

2.3.1. Sender's data structures

The following three main data structures, shown in Figure 2, are needed:

- A *hash table* with space for H words (where $H = nextPrime(2n)$) keeps in its component *word* the source word, in *posInVoc* the rank (or position) of the word in the ordered vocabulary, and in *freq* its frequency.
- *posInHT* is an n-element vector. $posInTH[i]$ points to the entry in the hash table that stores the ith most frequent word in the vocabulary.
- Array *top* contains F elements, where F is the maximum frequency. Each position implicitly represents a frequency value, that is, $top[f]$ is associated to words with frequency equal to f. For each possible frequency, vector *top* keeps a pointer to the entry in *posInHT* that points to the first (top) word with that frequency. If there are no words of frequency f_i, then $top[f_i]$ will point to the position of the first word j such that $f_j < f_i$.

A variable $zeroNode$ is also needed to indicate the first free position in the vocabulary, that is, the position in *posInHT* where the next new word will be inserted.

One concern is how to estimate F in a dynamic setup. It can be estimated heuristically using Heaps' Law [12]. Alternatively, in order to avoid vector *top* using up much more space than necessary, it can be implemented as a growing array that reallocates dynamically, doubling its size each time. It is also possible to substitute vector *top* by more sophisticated solutions [14].

To have an idea of the spaces involved, we present an example considering a text of 1 Gbyte from our experiments in Section 5. In this case, the highest frequency of a word is $F = 8,205,778$. Therefore the space requirements to keep vector *top* is $8,205,778 \times 4$ bytes ≈ 31 Mbytes, which is perfectly reasonable for current computers, although it can be reduced to ≈ 20 Mbytes with the aforementioned improvements.

Copyright © 2008 John Wiley & Sons, Ltd.
Prepared using speauth.cls

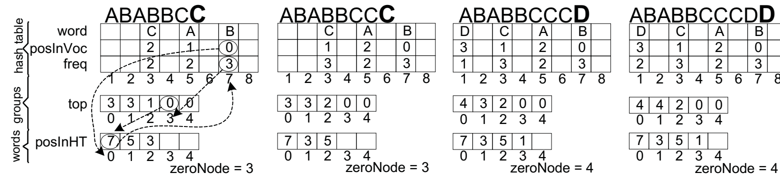


Figure 2. Transmission of words C, C, D and D having transmitted ABABBC earlier.

2.3.2. Receiver's data structures

The structures for the receiver are even simpler than those of the sender. The following three vectors are needed:

- A *word* vector that keeps the source words sorted by frequency. Its size is n.
- A *freq* vector that keeps the frequency of each word. That is, $freq[i] = f$, if the number of occurrences of the word stored in $word[i]$ is f. As the array *word*, this vector can keep up to n elements.
- Array *top*. As in the sender, this array gives, for each possible frequency, the word position of the first word with that frequency. It also has F positions.

The variable $zeroNode$ is also maintained by the receiver. The structures needed by the receiver are illustrated in Figure 3.

2.4. Sender and receiver processes

When the sender reads a word s_i, it uses the hash function to obtain its position p in the hash table, so that $hash(s_i) = p$ and therefore $word[p] = s_i$. After reading $f = freq[p]$, it increments $freq[p]$. The position of s_i in the vocabulary array is obtained as $i = posInVoc[p]$, so that codeword C_i is computed and sent. Now, word s_i must be promoted to the next block. For this sake, the sender algorithm finds the head of its block $j = top[f]$ and the corresponding position h of the word in the hash table $h = posInHT[j]$. Now, it is necessary to swap words i and j in vector $posInHT$. The swapping requires exchanging $posInHT[j] = h$ with $posInHT[i] = p$, setting $posInVoc[p] = j$ and $posInVoc[h] = i$. Once the swapping is done, j is promoted to the next block by setting $top[f] = j + 1$. If s_i turns out to be a new word, the sender will set $word[p] = s_i$, $freq[p] = 0$, and $posInVoc[p] = zeroNode$. Then the above procedure is followed with $f = 0$. Finally $zeroNode$ is also increased.

The receiver works very similarly to the sender, and it is even simpler. Its algorithm pseudo-code, plus the one of the sender, are shown in Figure 4. Figures 2 and 3 give an example of

Copyright © 2008 John Wiley & Sons, Ltd.
Prepared using speauth.cls

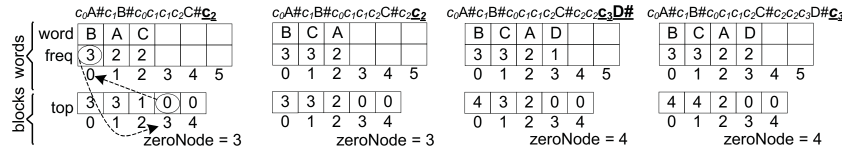


Figure 3. Reception of c_2, c_2, $c_3D\#$ and c_3 having received c_0A#c_1B#$c_0c_1c_1c_2$C# previously.

Sender main algorithm ( )
(1) Initialize *vocabulary structures*, *zeroNode* $\leftarrow$ 0;
(2) **for** $i \leftarrow 1$ **to** $n-1$ **do** $top[i] \leftarrow 0$;
(3) **for** *each new symbol* i **do**
(4) **read** s_i *from text*;
(5) $p \leftarrow f_{hash}(s_i)$;
(6) **if** $word[p] = Null$ ($s_i \notin word$) **then**
(7) $i \leftarrow zeroNode$;
(8) **send** $(encode(i))$;
(9) **send** s_i *in plain form*;
(10) **else**
(11) $i \leftarrow posInVoc[p]$;
(12) **send** $(encode(i))$;
(13) *update*();

Sender update ( )
(1) **if** $i = zeroNode$ **then** // new word
(2) $word[p] \leftarrow s_i$;
(3) $freq[p] \leftarrow 0$;
(4) $posInVoc[p] \leftarrow zeroNode$;
(5) $posInHT[zeroNode] \leftarrow p$;
(6) $zeroNode \leftarrow zeroNode + 1$;
(7) $f \leftarrow freq[p]$;
(8) $freq[p] \leftarrow freq[p] + 1$;
(9) $j \leftarrow top[f]$;
(10) $h \leftarrow posInHT[j]$;
(11) **swap** $(posInHT[i], posInHT[j])$;
(12) $posInVoc[p] \leftarrow j$;
(13) $posInVoc[h] \leftarrow i$;
(14) $top[f] \leftarrow j + 1$;

Receiver main algorithm ( )
(1) Initialize *vocabulary structures*, *zeroNode* $\leftarrow$ 0;
(2) **for** $i \leftarrow 1$ **to** $n-1$ **do** $top[i] \leftarrow 0$;
(3) **for** *each new codeword* C_i **do**
(4) $i \leftarrow decode(C_i)$;
(5) **if** $i = zeroNode$ **then**
(6) **receive** s_i *in plain form*;
(7) **output** s_i;
(8) **else**
(9) **output** $word[i]$;
(10) *update*();

Receiver update ( )
(1) **if** $i = zeroNode$ **then** // new word
(2) $word[i] \leftarrow s_i$;
(3) $freq[i] \leftarrow 0$;
(4) $zeroNode \leftarrow zeroNode + 1$;
(5) $f \leftarrow freq[i]$;
(6) $freq[i] \leftarrow freq[i] + 1$;
(7) $j \leftarrow top[f]$;
(8) **swap** $(freq[i], freq[j])$;
(9) **swap** $(word[i], word[j])$;
(10) $top[f] \leftarrow j + 1$;

Figure 4. Pseudo-code for sender and receiver processes in DETDC.

how the sender encodes the sequence of words ABABBCCDD and how the receiver decodes them.

Copyright © 2008 John Wiley & Sons, Ltd.
Prepared using speauth.cls

3. Dynamic (s, c)-Dense Codes

3.1. (s,c)-Dense Codes

End-Tagged Dense Code uses 2^{b-1} digits, from 0 to $2^{b-1}-1$, for the bytes that do not end a codeword (*continuers*), and the other 2^{b-1} digits, from 2^{b-1} to 2^b-1, for the last byte of the codeword (*stoppers*)[†]. Instead of using a fixed number of stoppers and continuers, (s, c)-Dense Code [6] adapts their number to the word frequency distribution in the corpus.

Definition 2. *Given source symbols with* nonincreasing *probabilities* $\{p_i\}_{0 \le i < n}$, *the corresponding* (s, c)*-Dense Code (SCDC) for the symbol in position* i *has* k *bytes* ($k \ge 1$), *for the* k *that satisfies:*

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

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

That is, using symbols of $b=8$ bits, the encoding process can be described as follows:

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

Example 3.1. The codes assigned to symbols $i \in 0 \ldots 15$ by a (2,3)-Dense Code are as follows: $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2{:}0 \rangle$, $\langle 2{:}1 \rangle$, $\langle 3{:}0 \rangle$, $\langle 3{:}1 \rangle$, $\langle 4{:}0 \rangle$, $\langle 4{:}1 \rangle$, $\langle 2{:}2{:}0 \rangle$, $\langle 2{:}2{:}1 \rangle$, $\langle 2{:}3{:}0 \rangle$, $\langle 2{:}3{:}1 \rangle$, $\langle 2{:}4{:}0 \rangle$, $\langle 2{:}4{:}1 \rangle$, $\langle 3{:}2{:}0 \rangle$, and $\langle 3{:}2{:}1 \rangle$. □

It is clear from Definition 2 that ETDC is a (2^{b-1},2^{b-1})-Dense Code and therefore SCDC is a generalization of ETDC that can obtain better compression by adjusting s and c to the text distribution. As in ETDC, the code does not depend on the exact symbol probabilities, just on their ordering by frequency.

The problem now consists of finding the s and c values (assuming a fixed b where $2^b = s + c$) that minimize the size of the compressed text for a specific word frequency distribution. A discussion on how to obtain the values that minimize the size of the compressed text for a specific word frequency distribution can be found in [6, 10].

The encoding and decoding algorithms are the same as those of ETDC, taking into account that s and c depend on the text (while with ETDC, both are always 128). Thus on-the-fly *encode* and *decode* algorithms are also available.

[†]For generality, we will keep considering bytes of b bits, not only 8.

Copyright © 2008 John Wiley & Sons, Ltd.

Prepared using speauth.cls

SCDC has only 0.2 percentage points of excess over the optimal PH code, improving upon ETDC by 0.7 percentage points and TH by 3.2 points. SCDC is simpler to build than Huffman Codes as well, and code generation is 45% faster than that of Huffman codes, although a little bit slower than ETDC (which is 60% faster than Huffman coding) because multiplications and divisions cannot be translated into faster bit shifts. To all these properties of SCDC, we have to add, as in the case of ETDC, all the search capabilities of TH.

As ETDC, SCDC has concepts in common with previous existing codes to compress integers. Golomb code [11] is a bit-oriented code, instead of byte-oriented, but it is also parameterized. Like SCDC, Golomb code has a parameter (sometimes called k), which is computed to best adapt the code to the distribution of the source symbols.

3.2. Towards Dynamic (s,c)-Dense Codes (DSCDC)

The main difference with respect to DETDC is that, at each step of the compression/decompression processes, it is mandatory not only to maintain the vocabulary sorted, but also to check whether the current value of s (and c) remains well tuned or if it should change.

The *update()* algorithm that maintains the list of words sorted by frequency is the same used in the case of DETDC. In addition, the test for a possible change of s has to be performed after calling this update process.

Both encoder and decoder start with $s = 256$. This s value is optimal for the first 255 words of the vocabulary, because it permits to encode all of them with just one byte. When the 256th word arrives, s has to be decreased by 1 since a two-byte codeword is needed. From this point on, s and c values are modified depending on the word frequency distribution.

We present next a heuristic technique to keep well tuned the values of s and c. Other heuristics that work well in most cases are described in [10].

3.3. Tuning the s and c values

The simplest approach to keep the s and c values well tuned as the compression/decompression progresses is based on comparing the size of the compressed text depending on the s and c values used to encode it.

The general idea is to compare the number of bytes that the compressed text, up to including word w_i, would occupy if it were encoded using $s-1$, s, and $s+1$. If that number becomes smaller by using either $s-1$ or $s+1$ instead of s, then the compressor switches to the new value of s from this point on. Therefore, in each step of the compression/decompression process, the value of s changes at most by one.

Three variables are needed: *prev*, *curr*, and *next*. Variable *prev* stores the size of the compressed text assuming that $s-1$ was used in the encoding/decoding process. In the same way, *curr* and *next* accumulate the size of the compressed text, assuming that it was encoded using the current s and $s+1$, respectively. At the beginning, the three variables are initialized to zero. Each time a word w_i is processed, *prev*, *curr*, and *next* are increased as follows: Let *countBytes(i)* be the function that computes the number of bytes needed to encode the ith word of the vocabulary. Then, the three variables are increased as follows:

Copyright © 2008 John Wiley & Sons, Ltd.

Prepared using speauth.cls

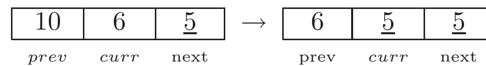
- $prev \leftarrow prev + countBytes(s-1, i)$
- $curr \leftarrow curr + countBytes(s, i)$
- $next \leftarrow next + countBytes(s+1, i)$

A change of the s value takes place either if $prev < curr$ or if $next < curr$. If $prev < curr$, then $s-1$ will become the new value of s ($s \leftarrow s-1$). On the other hand, if $next < curr$, then s will be increased ($s \leftarrow s+1$).

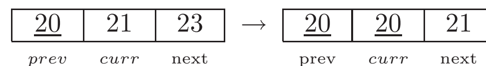
Therefore, we can easily decide in which direction s should be modified. Each time s changes, the values $prev$, $curr$, and $next$ are initialized again, and then the process continues. This initialization depends on the change of s that took place.

In order to keep the history of the process, we do not initialize the three values to zero, but we use the previous values. Of course, one of the three values (either $prev$ or $next$, depending on the direction of the change of s) is unknown and it is set to the same value of $curr$. That is:

- If s is increased then $prev \leftarrow curr$ and $curr \leftarrow next$ ($next$ does not change).



- If s is decreased then $next \leftarrow curr$ and $curr \leftarrow prev$ ($prev$ does not change).



There are other alternatives for this basic algorithm. For example, it would be possible to use an ε value as a threshold for the change in s. That is, the value of s would change only if $prev + \varepsilon < curr$ or $next + \varepsilon < curr$. In this way, fewer changes would take place, but in our experiments the differences were negligible.

Another possible choice would be to initialize the three variables $prev$, $curr$, and $next$ to zero when s changes. This choice would make the algorithm free from the previous history. This approach can be interesting in natural language documents where the vocabulary, and consequently its frequency distribution, changes frequently along the text. However, our experiments showed again that the differences in compression ratio were less than 0.01%.

Checking whether s and c should change is carried out by *CheckAndUpdateS()* algorithm. The pseudo-code of this algorithm and the one of *countBytes()* are shown in Figure 5. Notice that *countBytes()* is called at least twice in each execution of the *CheckAndUpdateS()* algorithm. The cost of *countBytes()* depends on the maximum codeword length, so its overall cost is proportional to the number of output symbols. This shows that s and c can be maintained well tuned without altering the overall complexity.

Figure 6 shows how the s value evolves in practice as compression progresses (the collections are described in Section 5). It can be seen that the dynamic encoder adapts the s value rapidly in order to reduce the codeword length. Therefore, the s value falls from 256 to 129 (recall that $b = 8$) when the first $16,512$ words are processed. When $n > 16,512$, three-byte codewords are needed, therefore the s value is increased. Fluctuations of s beyond that point depend on the word distribution.

Copyright © 2008 John Wiley & Sons, Ltd.
Prepared using speauth.cls

CheckAndUpdateS Algorithm $(s,\ c,\ i)$
(1) $prev \leftarrow$ **countBytes** $(s-1, i)$;
(2) $s_0 \leftarrow$ **countBytes** (s, i);
(3) **if** $prev < s_0$ **then**
(4) $\quad s \leftarrow s-1$; //s is decreased
(5) $\quad c \leftarrow c+1$;
(6) $\quad next \leftarrow s_0$;
(7) $\quad s_0 \leftarrow prev$;
(8) **else**
(9) $\quad next \leftarrow$ **countBytes** $(s+1, i)$;
(10) $\quad$ **if** $next < s_0$ **then**
(11) $\qquad s \leftarrow s+1$; //s is increased
(12) $\qquad c \leftarrow c-1$;
(13) $\qquad prev \leftarrow s_0$;
(14) $\qquad s_0 \leftarrow next$;

countBytes Algorithm $(s_i,\ i_{pos})$
(1) $k \leftarrow 1$;
(2) $last \leftarrow s_i$;
(3) $pow \leftarrow s_i$;
(4) $c_i \leftarrow 256 - s_i$;
(5) **while** $last \leq i_{pos}$ **do**
(6) $\quad pow \leftarrow pow \times c_i$;
(7) $\quad last \leftarrow last + pow$;
(8) $\quad k \leftarrow k+1$;

Figure 5. countBytes and CheckAndUpdateS algorithms.

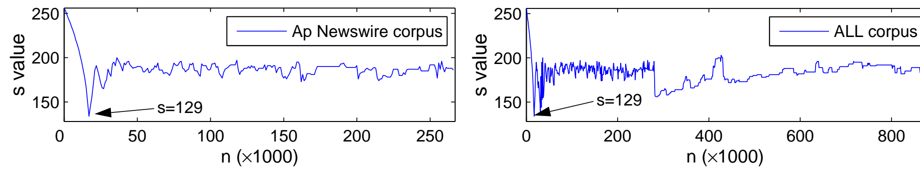


Figure 6. Evolution of s as the vocabulary grows.

4. Block-wise Versions of Dense Codes

A natural choice to cope with the real-time scenario is to cut the text into *blocks* that can be compressed separately using a semi-static compressor. This solution is simple and likely to provide efficient decompression and searching. The idea behind this technique is, on the one hand, that it can adapt better to different distributions on different parts of the text, and on the other hand, that the codewords used in each block are shorter on average, since there are fewer different words in each block than in the whole text.

However, it is not obvious whether or not this is a good idea. If the blocks have to be too large to provide good compression, due to the burden of storing the local vocabulary, the real-time nature of the scheme might be questionable. We performed several studies in this line, and we found that the simple approach of cutting the text into blocks, and compressing them with a semistatic approach, does not obtain good results, as it can be seen in Table I (the collections are described in Section 5). As shown, using a simple block-wise compressor represents a severe restriction for real-time transmission, since to achieve competitive compression ratios

Copyright © 2008 John Wiley & Sons, Ltd.

Table I. Compression ratio (in percentage) of a semistatic block-wise ETDC with different block sizes (in Mbytes) vs ETDC and DETDC.

	0.5 Mb	1 Mb	2 Mb	5 Mb	10 Mb	15 Mb	ETDC	DETDC
CR	39.26	36.80	35.21	33.50	32.58	32.28	31.94	31.99
AP	42.07	39.35	37.45	35.60	34.57	34.12	32.90	32.91
ALL	41.53	39.04	37.33	35.55	34.58	34.13	33.66	33.66

the sender should have to delay the transmission of the compressed text until the available text reaches a considerable size (recall the scenarios depicted in Section 1).

In [9], some techniques to store the vocabulary of each block in the form of a short prelude were shown. However, in this work it is assumed that the sender and the receiver share the entire vocabulary, and therefore the prelude only provides information about which words, from the general vocabulary, are present in the block, in addition to some information needed for the encoding. Yet, in the case of a dynamic scenario, it is necessary to send new words (or separators), which were not known until then, as compression progreses. Therefore further development is required to profit from this research line.

5. Experimental Results

We used a large text collection from TREC-2[‡], namely AP Newswire 1988 (AP), as well as from TREC-4, namely Congressional Record 1993 (CR). As a small collection we used the Calgary corpus[§] (CALGARY). We created two larger corpora ALL_FT and ALL by aggregating several texts from TREC-2, TREC-4 and the Calgary corpus. We used the spaceless word model [17] 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 empirically compared the compression ratio and compression/decompression speed of DETDC and DSCDC against our own implementation of a dynamic word-based byte-oriented Plain Huffman (DPH)[¶], *Gzip*[‖], a Ziv-Lempel compressor with performance very similar to *zip* [24], *Bzip2*[**], a block sorting compressor [7] and an *arithmetic encoder* coupled with a word-based modeler [8]. Finally, as a baseline, we included dynamic (*DVyte*) and semistatic (*SVbyte*) implementations of a variable byte code [9, 21]. *DVbyte* and *SVbyte* do not use any

‡ http://trec.nist.gov

§ ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus.

¶ Open-source implementations of DETDC, DSCDC and DPH, as well as pseudo-codes for their compression and decompression processes, are available at http://rosalia.dc.fi.udc.es/codes/.

‖ http://www.gzip.org/.

** http://www.bzip.org/.

Copyright © 2008 John Wiley & Sons, Ltd.

statistical information of the text in order to obtain a good compression ratio. They just assign the first codeword to the first word in the text, the second codeword to the second distinct word in the text, and so on. As shown later, this yields worse compression ratio than ETDC and DETDC, but better compression times. Tables show the *default compression* setting of *Gzip*. ETDC, SCDC, and PH, the semistatic counterparts of DETDC, DSCDC, and DPH respectively, were also included in the comparisons to discuss the effects of dynamism.

An isolated Intel®Pentium®-IV 3.00 GHz system (16Kb L1 + 1024Kb L2 cache), with 4 GB dual-channel DDR-400Mhz RAM was used in our tests. It ran Debian GNU/Linux (kernel version 2.4.27). The compiler used was gcc version 3.3.5 and `-O9` compiler optimizations were set. Time results measure CPU user time in seconds.

5.1. Compression ratio, compression and decompression time

Table II (a) shows the compression ratios obtained when compressing the different corpora. As expected, *Bzip2* yields the best compression ratio. *Gzip* obtains the worst compression ratio in medium size texts, while in the shortest and largest texts, both versions of the *Vbyte* code are the worst ones. Arithmetic compression also obtains good results (1-2 percentage points over *Bzip2*). Our techniques compress more than *Gzip* except in small collections (CALGARY), where the vocabulary size is still significant compared to the text size. DPH, which generates optimal prefix-free codes, overcomes DSCDC by less than 0.3 percentage points, and DETDC loses around 0.6 percentage points with respect to DSCDC. It is important to note the gain in compression achieved by DETDC and ETDC with respect to *DVbyte* and *SVbyte* respectively; the use of a statistical model results in 3-11 percentage points of improvement, depending on the size of the text. Finally, it is also interesting to point out that adding dynamism to PH, ETDC, and SCDC involves only a slightly loss of compression ratio. In most cases, such a loss is less than 0.1 percentage points.

Table II (b) shows compression times. As expected, *DVbyte* is the fastest alternative, since it does not compute statistics of the source text nor performs vocabulary permutations during the compression process. Yet, DETDC is also very fast, obtaining much better compression.

DSCDC is slightly slower than DETDC due to the need of maintaining parameters s and c well tuned. DPH also obtains good performance, but it is overcome by the dense compressors because of the complexity of dynamically maintaining a well-formed byte-oriented Huffman tree. *Bzip2*, *Arith*, and *Gzip* are significantly slower than the rest of the techniques.

Comparing the dynamic techniques against their semistatic counterparts (PH, ETDC, SCDC and *SVbyte*), it can be observed that performing only one pass over the text to compress makes dynamic techniques faster. In fact, DETDC is around 30% faster than ETDC, whereas DSCDC is around 20% faster than SCDC. DPH is also able to overcome PH in compression speed. However, since DPH might have to update a higher Huffman tree for each source word, gaps (in compression speed) between DPH and PH decrease as the size of the collection grows. The case of *SVbyte* is even worse, losing around 50% of speed, since it wastes its advantage (it does not have to sort the vocabulary) writing a bigger output file.

Table II (c) shows decompression times. We remark that *Gzip* is regarded as a very efficient technique for decoding. However, dynamic dense codes still obtain good times. DETDC has a decompression performance similar to that of *Gzip* (except in corpus ALL). DSCDC pays the

Copyright © 2008 John Wiley & Sons, Ltd.

Table II. Compression Ratio (in percentage) and, compression and decompression times (in seconds).

(a) Compression ratio

Corpus	Size KB	Gzip	DPH	DETDC	DSCDC	DVbyte	PH	ETDC	SCDC	SVbyte	Arith	Bzip2
CALGARY	2,081	36.95	46.55	47.73	46.81	55.18	46.24	47.40	46.61	53.16	34.68	28.92
CR	49,888	33.29	31.10	31.99	31.33	34.83	31.06	31.94	31.29	34.83	26.30	24.14
AP	244,760	37.32	32.09	32.91	32.36	36.96	32.07	32.90	32.35	36.95	27.94	27.25
ALL_FT	577,704	34.94	31.71	32.54	31.85	41.59	31.70	32.53	31.84	41.59	27.85	25.87
ALL	1,055,391	35.09	32.85	33.66	33.03	45.00	32.83	33.66	33.02	45.00	27.98	25.98

(b) Compression time

Corpus	Gzip	DPH	DETDC	DSCDC	DVbyte	PH	ETDC	SCDC	SVbyte	Arith	Bzip2
CALGARY	0.34	0.12	0.09	0.11	0.07	0.16	0.16	0.16	0.15	0.41	0.70
CR	7.47	2.78	2.16	2.41	1.66	3.07	3.07	3.05	2.92	7.62	17.58
AP	39.09	15.13	11.91	13.39	8.791	16.21	16.55	16.38	16.05	39.65	85.69
ALL_FT	85.05	35.79	28.20	31.52	20.44	37.90	39.27	38.70	38.25	93.58	208.58
ALL	160.01	71.54	55.31	61.35	39.55	72.77	75.58	75.20	73.78	171.56	375.55

(c) Decompression time

Corpus	Gzip	DPH	DETDC	DSCDC	DVbyte	PH	ETDC	SCDC	SVbyte	Arith	Bzip2
CALGARY	0.04	0.07	0.04	0.05	0.03	0.04	0.03	0.04	0.03	0.38	0.30
CR	0.94	1.83	0.94	1.10	0.63	0.62	0.59	0.67	0.62	6.57	7.14
AP	5.19	10.24	5.27	6.10	3.67	3.29	3.34	3.46	3.44	33.89	37.75
ALL_FT	11.38	24.01	12.63	14.56	8.52	7.60	7.54	7.96	8.13	80.29	85.11
ALL	21.05	50.82	25.27	28.62	17.45	14.26	14.56	15.08	16.96	147.15	156.18

extra cost of maintaining the values of s and c tuned (as well as a slightly slower decoding algorithm) and is overcome by DETDC by around 10%. DPH, due to its complex update algorithm, is about two times slower than DETDC and DSCDC. Finally, *DVbyte* is faster than *Gzip*, DETDC and DSCDC. More precisely, it is around 40% faster than DETDC, again due to its simplicity.

As expected [6], the semistatic techniques obtain the best decompression times. The arithmetic compressor and *Bzip2* are by far the slowest techniques.

To sum up, DETDC is easier to program, compresses more and faster than *Gzip*, being also very fast at decompression. Yet, DETDC requires more memory than *Gzip*. We used the ALL corpus to test the memory consumption of DETDC and DSCDC compared to that of *Gzip*. In compression, DETDC and DSCDC consume around 90 Mbytes, whereas Gzip only consumes 720 Kbytes. Decompression depicts a similar situation: dense compressors consume 60 Mbytes and *Gzip* uses 520 Kbytes. Note, however, that *Gzip* cannot profit from using more memory, as that could only be used to enlarge the window size, and this is chosen to be generally optimal (a longer window requires longer pointers in the compressed text). Thus there is no really a

Copyright © 2008 John Wiley & Sons, Ltd.

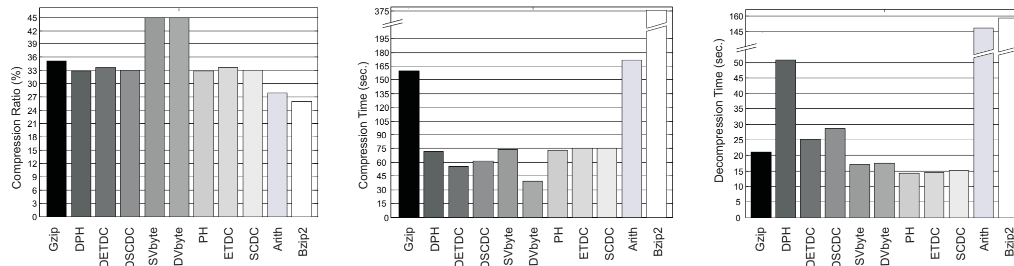


Figure 7. Compression ratio (left), compression time (middle) and decompression time (right).

compression ratio versus memory space tradeoff. In current computers, the space required by the dense coders is perfectly affordable.

The good features inherited from ETDC make DETDC an interesting choice for dynamic compression of natural language texts. DSCDC is also a good alternative to *Gzip*. It compresses faster than *Gzip* and its compression ratio is much better. *DVbyte* is a very fast alternative, but its compression ratios make DETDC a better choice to obtain a good balance between space and time. An overall comparison among all the discussed compression techniques is given in Figure 7.

6. Searching compressed and uncompressed text

We performed multi-pattern searches for randomly chosen patterns over both the compressed and uncompressed versions of collection ALL. We present results for four search algorithms that work over compressed text and four well-known algorithms for searching plain text.

The first technique works over text compressed with ETDC and SCDC. We use our own implementation of *Set-Horspool* algorithm [13, 18], with the small modification needed to deal with ETDC and SCDC [6, 10] (namely, it is necessary to verify that the character that precedes an occurrence is actually a stopper, as in a dense code a codeword can be a suffix of a longer codeword). In this case, the search patterns are first encoded and then directly searched for in the compressed text. Results regarding searches over text compressed with PH are not included here as they are known to be much worse than those on ETDC and SCDC [6].

On the other hand, we consider searches over text compressed with DETDC. Since the codewords generated by DETDC might vary each time a source word is input, a Boyer-Moore type search is not suitable. In practice, searching text compressed with DETDC (and also with DSCDC) consists in simulating the decompression process (just without emitting the source words), so that all bytes in the compressed file are processed. This is the reason why we call it *all-bytes*. Basically, the searcher processes the whole file one codeword at a time keeping

Copyright © 2008 John Wiley & Sons, Ltd.

track of the codewords associated to the searched patterns along the compressed text, and reporting their occurrences. In DETDC, the searcher might only be interested in counting the occurrences of the patterns (for example to classify documents) or might be interested in displaying an uncompressed context around each occurrence. If local decompression is needed, the searcher must not only search for the patterns, but also be able to rebuild the vocabulary of the decompressor. This variant is marked as *all-bytes + dec* in the experiments. We do not include search times for DSCDC nor for DPH since, just as they are slower than DETDC at decompression, they also obtain worse results at searches.

The fourth search tool included in our comparison is author's implementation of *LZgrep* [19]. *LZgrep* permits searching text compressed with LZ77/LZ78/LZW formats [24, 25] faster than performing decompression plus searching. In our experiments, we were aiming at using the best alternative for *decompression and searching*. Since LZ77 is the fastest Ziv-Lempel variant at decompression, applying *LZgrep* over text compressed with a LZ77-based technique such as *Gzip*, is the fastest choice. Therefore, *LZgrep* was run over text compressed with *Gzip -9* [††].

Four different algorithms were tested to search the uncompressed text: *i)* our own implementation of *Set-Horspool* algorithm, *ii)* author's implementation of *Set Backward Oracle Matching algorithm* (SBOM) [1], *iii)* author's implementation of *Simplified Set Backward Oracle Matching algorithm* (SSBOM) [18] and *iv)* the *agrep*[‡‡] software [23, 22], a fast pattern-matching tool which allows, among other things, searching a text for multiple patterns. *Agrep* searches the text and returns those chunks containing one or more search patterns. The default chunk is a line, and the default chunk-separator is the *newline* character. Once the first search pattern is found in a chunk, *agrep* skips processing the remaining bytes in the chunk. This speeds up *agrep* searches when a large number of patterns is sought. However, it does not give the exact positions or counts of the search patterns. To make a fairer comparison, in our experiments, we also tried *agrep* with the *reversed* patterns, which are less likely to be found. This maintains essentially the same statistics of the searched patterns and reflects better the real search cost of *agrep*.

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

To choose the search patterns, we considered the vocabulary associated to corpus ALL. From that vocabulary, we skipped both the *stopwords* (prepositions, articles, etc.) and the separators (sequences of non-alphanumerical characters), as these are almost never search targets. Yet, with the aim of avoiding the search for misspellings, we also skipped words appearing only once in the text. As a result, we obtained a *list of candidate patterns*. Then, following the model [17] where each vocabulary word was sought with uniform probability, we extracted 100 sets with K words of length L at random from the list of candidate patterns. As it is shown in Table III, we consider lengths $L = 5$, 10, and > 10, and each set can consist of $K = 5$, 10, 35, 50, 100, 200, 400, and 1000 patterns. Therefore, for each pair (L_i, K_j), the values shown in Table III

[††]Searching or decompressing text compressed with *Gzip* is more efficient as less data has to be processed. Therefore, using *Gzip -9* at compression yields the best search/decompression times.
[‡‡]ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z.

Copyright © 2008 John Wiley & Sons, Ltd.

Table III. Multi-pattern search times over corpus ALL (in seconds).

Search type	length of pattern	number of patterns 5	10	25	50	100	200	400	1000
ETDC Set-Horspool	5	0.645	0.764	1.158	1.807	2.497	3.223	3.485	3.902
	10	0.681	0.794	1.137	1.732	2.470	3.148	3.393	3.732
	> 10	0.657	0.759	1.151	1.750	2.488	3.164	3.419	3.746
SCDC Set-Horspool	5	0.609	0.701	1.017	1.584	2.283	3.289	3.826	4.528
	10	0.638	0.726	1.017	1.535	2.299	3.195	3.675	4.031
	> 10	0.615	0.702	0.993	1.522	2.282	3.151	3.598	4.067
DETDC all bytes	5	10.484	10.541	10.557	10.471	10.667	10.557	10.643	10.660
	10	10.501	10.541	10.512	10.535	10.707	10.539	10.634	10.693
	> 10	10.521	10.534	10.624	10.507	10.688	10.573	10.625	10.734
DETDC + dec all bytes	5	14.559	14.495	14.516	14.525	14.602	14.574	14.579	14.619
	10	14.476	14.551	14.521	14.511	14.664	14.550	14.586	14.669
	> 10	14.529	14.616	14.509	14.522	14.677	14.560	14.603	14.662
LZgrep -s -c	5	15.129	15.085	15.200	15.066	15.139	–	–	–
	10	15.171	15.120	15.201	15.083	15.162	–	–	–
	> 10	15.121	15.168	15.182	15.176	15.130	–	–	–
Agrep -s -c *default*	5	5.488	5.348	4.523	3.641	2.868	1.195	0.486	0.212
	10	2.897	3.123	3.644	3.883	3.193	2.480	1.493	0.730
	> 10	2.926	3.145	3.869	3.883	3.573	2.827	1.890	0.986
Agrep -s -c *rev. patterns*	5	5.706	5.838	6.426	6.999	10.490	9.799	8.413	5.529
	10	2.824	3.019	3.530	4.620	4.764	5.031	5.477	6.478
	> 10	2.832	3.039	3.619	4.641	4.777	5.070	5.570	6.786
Set-Horspool	5	2.065	3.194	5.491	7.469	9.143	11.138	13.262	16.456
	10	1.913	2.992	4.566	5.712	6.724	8.030	9.633	12.847
	> 10	1.921	2.975	4.677	5.748	6.815	8.070	9.644	13.067
SBOM	5	3.796	4.784	6.020	8.174	10.934	13.290	15.996	21.688
	10	2.803	3.491	5.238	6.796	8.213	9.924	9.924	12.373
	> 10	2.847	3.611	5.384	6.883	8.201	10.051	12.616	17.372
SSBOM	5	4.106	5.161	6.432	8.574	11.106	13.544	16.680	23.102
	10	2.902	3.503	5.046	6.405	7.684	9.447	12.117	17.369
	> 10	2.935	3.623	5.160	6.459	7.726	9.617	12.373	18.103

give the average time needed to perform 100 searches (using the same 100 sets of preselected K_j search patterns) with each of the search techniques compared. Figure 8 summarizes the results obtained by searching for K_j patterns of 5 bytes, in graphical form. This is less precise but easier to visualize.

Results show that searches over text compressed with DETDC can be done much faster than decompressing plus searching. This is actually an interesting property [19] for a dynamic compressor. Moreover, *all-bytes + dec* obtains slightly better search times than *LZgrep*, its main competitor in a dynamic scenario.

Of course, searches over text compressed with a dynamic compressor cannot compete against searches over text compressed with a semistatic compressor, being also usually slower than just searching the uncompressed text when a few patterns are sought.

As expected, searching text compressed with ETDC and SCDC is much faster than searching the uncompressed text, and around 3-4 times faster than searching text compressed with DETDC. Only *default agrep* (which skips lines where patterns are found) can overcome the searches over ETDC and SCDC, yet this occurs when more than 100 words are searched for and, as explained, does not reflect the real cost of a search. Closer to the real search cost of *agrep* is its *reversed patterns* version, except when many short patterns are sought, so that

Copyright © 2008 John Wiley & Sons, Ltd.

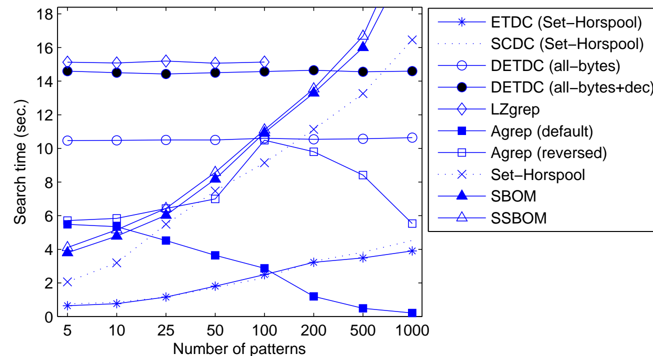


Figure 8. Searching for a variable number of patterns of 5 letters.

they are usually found in the text even in reverse form. Algorithms SBOM and SSBOM are faster than *agrep* when the number of patterns is small, yet they become slower as the number of patterns increases.

As expected, the use of longer patterns improves the search speed in the uncompressed text. However, this has little effect in the search time over text compressed with ETDC and SCDC, as in this case the searcher usually looks for codes of 2–3 bytes. In general, one-byte codes are rarely searched for when we search for less than 400 patterns (because there are few of them, and they usually correspond to stopwords). In the case of searches over text compressed with DETDC, search times are independent of the length of the patterns, as they only depend on the number of codewords in the compressed file.

If we focus on the search algorithms based on *Set-Horspool*, we realize that a larger number of patterns favors the search on the compressed over the uncompressed text. The main reason is that Horspool's algorithm benefits from a lower probability of two characters (from the text and the pattern) being equal. The lower this probability, the more patterns can be handled efficiently. In the compressed version (using ETDC) of the corpus ALL, this probability is $1/119.4 \approx 0.008$, whereas in the plain version, it is $1/19.3 \approx 0.052$.

As it was introduced above, searching text compressed with DETDC can be done more efficiently than decompressing plus searching, as it happens in *LZgrep*, but not as fast as just searching the uncompressed version of the text. However, when a large number of patterns ($>$100) are sought, searching DETDC becomes faster than searching the uncompressed text. This occurs because the simple *all-bytes* searcher is almost independent of the number of search patterns. The *all-bytes + dec* searcher that works on DETDC is around 40% slower than the *all-bytes* variant. Those gaps are the result of having to perform the whole *update* process of the vocabulary for each codeword that appears in the compressed text, instead of just keeping track of the positions of the searched patterns in the vocabulary. As in DETDC, the results

Copyright © 2008 John Wiley & Sons, Ltd.
Prepared using speauth.cls

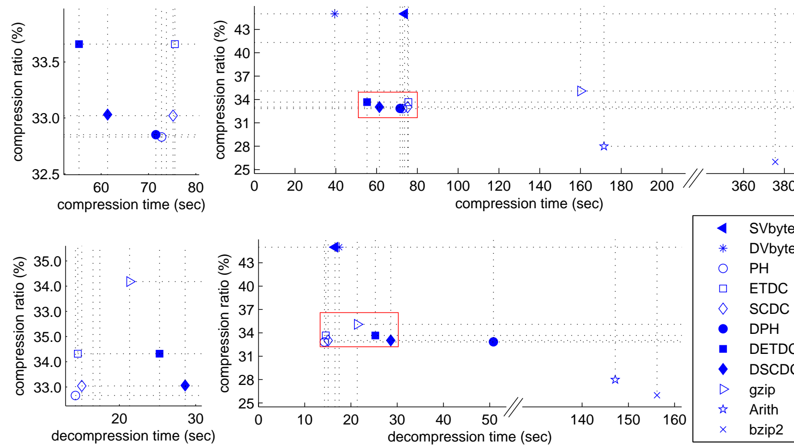


Figure 9. Space/time tradeoffs among dynamic techniques, on corpus ALL, related to compression/decompression time.

obtained with *LZgrep* are almost independent of the number of searched patterns and their length. Comparing DETDC against *LZgrep*, we found that *LZgrep* is around 40-50% and 5-8% slower than *all bytes* and *all bytes + dec* searchers, respectively.

7. Conclusions

We have addressed the problem of efficient transmission of natural language text documents. This was done by adding dynamism to two existing word-based byte-oriented semistatic compressors such as ETDC, and SCDC. They obtain compression ratios slightly worse than those of their semistatic counterparts, but better compression times.

More precisely, DETDC and DSCDC enjoy several desirable features: full real-time transmission, simplicity, good compression ratios (around 31-34%), fast compression and decompression, and the ability to search the compressed text without decompressing (searching simulates decompression but it is faster as the searcher does not have to output the text). The new compressors stand out as attractive space/time trade-offs within the current state of the art, and have the additional benefit of being very simple to program.

In Figure 9, we focus on corpus ALL, showing the trade-off between compression ratio and compression and decompression speed for most of the compressors used in our experiments. The left part of each graphic shows an enlargement of the clump of values (within a rectangle) that appears in the main plot.

Copyright © 2008 John Wiley & Sons, Ltd.

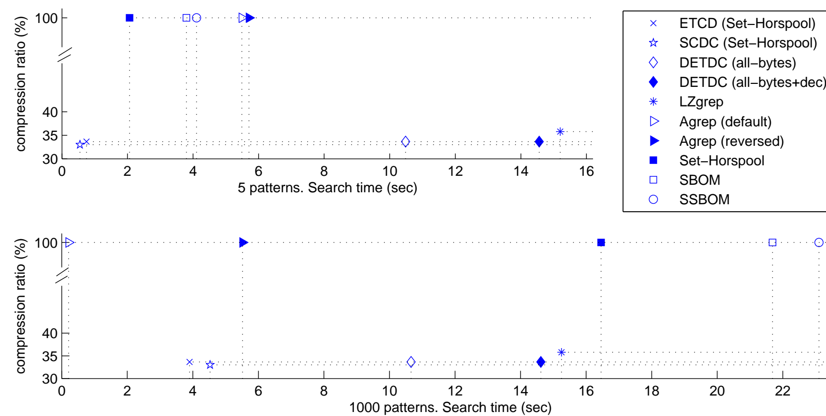


Figure 10. Space/time tradeoffs on corpus ALL related to compression/search time when patterns of 5 letters are used.

Our empirical results showed that searches performed over text compressed with DETDC obtain better results than those obtained with *LZgrep*, and therefore, they are faster than just decompressing plus searching. Moreover, the search times obtained are independent of the length and number of searched patterns. This fact implies that the search over text compressed with dynamic dense codes is faster than well-known techniques that work over uncompressed text when many patterns are searched for (>100). Figure 10 shows the trade-off between compression ratio and search time for all the search tools used in our experiments assuming that either 5 or 1000 patterns of 5 bytes are searched. Notice that *agrep* search times are unfair when 1000 patterns are searched for, as it explained in Section 6. Indeed, already for 100 patterns, *agrep* times reach those of our techniques, and they would keep worsening against ours, if the artifacts of its line-counting approach were deactivated.

As future work we are interested in the development of new dynamic codes that permit us to search the compressed text more efficiently. We are now targeting at improving the promising preliminary results obtained, in both decompression and searches, by Dynamic Lightweight ETDC (DLETDC), an asymmetric version of DETDC presented in [5], and to extend the result to SCDC. In these two asymmetric techniques, some loss of compression effectiveness and compression speed (with respect to DETDC and DSCDC) is permitted in order to improve decompression time and mainly search capabilities. In addition, we aim to develop these compressors to be used in low-computational power devices such as PDAs or mobile phones, where restrictions of memory can be found as well.

REFERENCES

Copyright © 2008 John Wiley & Sons, Ltd.
Prepared using speauth.cls

1. C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In *Proceedings of the 26th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'99)*, LNCS 1725, pages 291–306. Springer-Verlag, 1999.
2. J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.
3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
4. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Simple, fast, and efficient natural language adaptive compression. In *Proceedings of the 11th International Symposium on String Processing and Information Retrieval (SPIRE'04)*, LNCS 3246, pages 230–241. Springer-Verlag, 2004.
5. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Efficiently decodable and searchable natural language adaptive compression. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'05)*, pages 234–241, New York City, 2005. ACM Press.
6. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.
7. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
8. J. Carpinelli, A. Moffat, R. Neal, W. Salamonsen, L. Stuiver, A. Turpin, and I. Witten. Word, character, integer, and bit based compression using arithmetic coding. Relevant software available at http://www.cs.mu.oz.au/~alistair/arith_coder/, 1999.
9. J.S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE'05)*, LNCS 3772, pages 1–12. Springer-Verlag, 2005.
10. A. Fariña. *New Compression Codes for Text Databases*. PhD thesis, Database Laboratory, University of A Coruña, Spain, 2005. Available at http://coba.dc.fi.udc.es/~fari/phd/.
11. S. W. Golomb. Run-length encodings. *IEEE Trans. Inform. Theory*, IT-12:399–401, 1966.
12. H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, New York, 1978.
13. R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.
14. D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163–180, 1985.
15. A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.
16. A. Moffat and Turpin A. *Compression and Coding Algorithms*. Kluwer_Academic Publishers, 2002.
17. E. 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.
18. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
19. G. Navarro and J. Tarhio. LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text. *Software Practice and Experience (SPE)*, 35(12):1107–1130, 2005. Relevant software available at http://www.dcc.uchile.cl/~gnavarro/software/lzgrep.tar.gz.
20. A. Turpin and A. Moffat. Fast file search using text compression. In *Proceedings of the 20th Australian Computer Science Conference (ACSC'97)*, pages 1–8, 1997.
21. H. E. Williams and J. Zobel. Compressing integers for fast file access. *COMPJ: The Computer Journal*, 42(3):193–201, 1999.
22. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 153–162, 1992.
23. S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.
24. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
25. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

Copyright © 2008 John Wiley & Sons, Ltd.

Prepared using speauth.cls