# Dynamic Lightweight Text Compression

NIEVES BRISABOA, ANTONIO FARIÑA
University of A Coruña, Spain
and
GONZALO NAVARRO
University of Chile, Chile
and
JOSÉ PARAMÁ
University of A Coruña, Spain

We address the problem of adaptive compression of natural language text, considering the case where the receiver is much less powerful than the sender, as in mobile applications. Our techniques achieve compression ratios around 32% and require very little effort from the receiver. Furthermore, the receiver is not only lighter, but it can also search the compressed text with less work than the necessary to uncompress it. This is a novelty in two senses: it breaks the usual compressor/decompressor symmetry typical of adaptive schemes, and it contradicts the long-standing assumption that only semistatic codes could be searched more efficiently than the uncompressed text. Our novel compression methods are in several aspects preferable over the existing adaptive and semistatic compressors for natural language texts.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data Compaction and Compression; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms: Text Compression, Searching Compressed Texts
Additional Key Words and Phrases: Adaptive natural language text compression, Real time transmission, Compressed pattern matching

## 1. INTRODUCTION

Research in the last decade has shown that text compression and retrieval are intimately related. Text compression not only serves to save space, processing, and

Funded in part by AECI PCI A/8065/07, (for the Spanish group) by MEC (PGE and FEDER) grant (TIN2006-15071-C03-03), Xunta de Galicia grant (ref. 2006/4) and (for the third author) by Millennium Nucleus Center for Web Research, grant (P04-067-F), Mideplan, Chile, and Fondecyt Grant 1-080019, Chile.
A preliminary partial version on this work appeared in *Proc. SIGIR'05* [Brisaboa et al. 2005b].
Authors' address: Nieves Brisaboa , Antonio Fariña , José Paramá , Department of Computer Science, University of A Coruña, Facultade de Informática, Campus de Elviña, s/n 15071 A Coruña, Spain. {brisaboa, fari, parama}@udc.es. Gonzalo Navarro, Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl.

transmission time, but is also an extremely effective device to speed up search and retrieval operations in natural language text collections [Ziviani et al. 2000; Navarro et al. 2000].

Text compression is also extremely useful in data transmission. When all the data to be sent can be compressed before the beginning of the transmission, statistical *two-pass* techniques, also called *semistatic*, can be used. These techniques perform a first pass over the text to gather the list of source symbols and their frequency. This information is used to construct a *model*[1] of the text, which is used by the *encoder* to compute the codeword corresponding to each source symbol. Then, in a second pass over the original text, source symbols are replaced by their codewords. The sender must transmit the model along with the compressed text to inform the receiver about the correspondence between source symbols and codewords.

This approach is not suitable for *real-time* transmission scenarios, where the sender should be able to start the transmission of the compressed text without preprocessing the whole text, and the receiver should be able to start the decompression as soon as it starts to receive the compressed text.

*Dynamic* or *adaptive* compression techniques are designed to deal with the real-time transmission scenario. These techniques perform only one pass over the text (so they are also called *one-pass*) and are able to start the compression and transmission as they read the text. In this case, the model is not sent, as the receiver can infer it during the decompression of the received data.

Recently, statistical semistatic compression techniques especially designed for natural language text have shown that it is possible to search the compressed text much faster (up to 8 times) than the original text, obtaining at the same time very appealing compression ratios[2], around 25%-35%. The key idea of these techniques is to regard the text to be compressed as a sequence of words instead of characters [Bentley et al. 1986]. A Huffman-based compressor using this approach was presented in [Moffat 1989]. In the same line, Moura et al. [2000] introduced two coding methods called *Plain Huffman (PH)* and *Tagged Huffman (TH)*. The byte-oriented Plain Huffman achieves compression ratios close to 30%, as opposed to the 25% that is achieved by using bit-oriented codes [Turpin and Moffat 1997b]. In exchange, decompression is much faster because bit manipulations are not necessary. Tagged Huffman provides an improved self-synchronization ability that allows fast direct search of the compressed text and local decompression of text passages. This improvement has a penalty in compression ratio, which worsens to around 34%.

The newly proposed family of coding methods called *dense codes* has been shown to be preferable in most aspects to Huffman coding for natural language [Brisaboa et al. 2007]. 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 compressor based on dense codes, which is called *End-Tagged Dense Code (ETDC)*, achieves compression ratios around 31% using a semistatic zero-order word-based model. An improved variant, $(s, c)$*-Dense Code (SCDC)*, reaches less than 0.3 percentage points over PH compression ratio. An-

[1] In this paper, we consider *zero-order* models, which provide the probability of each source symbol without taking into account the previous symbols in the source stream.
[2] The size of the compressed text as a percentage of its original size.

other competitive semistatic proposal is the *Restricted Prefix Byte Code* [Culpepper and Moffat 2005], which compresses natural text about 1% better than SCDC, in exchange for usually slower searching and no self-synchronization.

Adaptive compression methods based on the Ziv-Lempel family [Ziv and Lempel 1977; 1978] (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. Among the adaptive compressors, dynamic arithmetic coding over PPM-like modelling [Cleary and Witten 1984] obtains compression ratios around 24%, but it requires significant computational effort by both the sender and the receiver, being quite slow at both ends.

A seemingly easy way to achieve better compression ratios for natural language texts in real-time scenarios is to process the input by chunks, and use semi-static compression for each chunk. This, however, does not work well, because it is necessary to use chunks of at least 5-10 Mbytes to compensate for the extra space used by the model (the vocabulary of words) [Moura et al. 2000; Brisaboa et al. 2008]. The burden of sending many vocabularies could be alleviated by sending only the differences with respect to the previous one (that is, the swaps to make in the list of words sorted by frequency). Still, using chunks of 5-10 Mbytes is unapplicable in several real-time applications, such as in those where two parties exchange many short messages along a session, so that the overall exchanged text is large enough to allow for compression, but transmission of short messages must be carried out immediately (e.g. chats, transactions, browsing scenarios, broadcasting of news, etc.). To cope with those cases, one could use a much smaller chunk size, profiting from the cheaper differential vocabulary transmission.

Improving upon this idea, Brisaboa et al. [2008] introduce two truly adaptive compressors based on dense codes, called *Dynamic ETDC (DETDC)* and *Dynamic SCDC (DSCDC)*. As in general adaptive compression, no model information is transmitted at all. Both DETDC and DSCDC turn out to be interesting alternatives: with compression ratios around 31-34%, they are as fast as the fastest Ziv-Lempel compressors (which achieve only 40% compression ratio) and about 10 times faster than dynamic PPM. In decompression, dynamic dense codes have a performance similar to that of Ziv-Lempel compressors and are 25 times faster than dynamic PPM. Finally, searches performed over text compressed with DETDC obtain better results than those obtained by *LZgrep* [Navarro and Tarhio 2005], yet far from searching texts compressed with semistatic methods. Hence, dynamic dense codes present several advantages over state-of-the-art adaptive compressors.

However, this adaptive compression retains some important drawbacks. In statistical adaptive compression, the model changes each time a text word is processed. This has two important consequences. First, the receiver has to keep track of all these changes in order to maintain its model updated, implying a significative computational effort. This is especially unfortunate in cases where the receiver has limited computational power, such as in mobile applications. Second, these frequent changes of the model make it difficult to carry out direct searches, as the search pattern looks different in different points of the compressed text.

It is not difficult to find adaptive compression scenarios where a direct search on the compressed text (i.e., without decompressing it) can be a valuable tool.

For example, consider mobile environments where stations broadcast information (like traffic, places of interest, etc.) continuously to devices in their cells. It is likely that receivers are not interested in decompressing all of the information they receive. So they would search the arriving compressed text for some keywords that describe topics of interest. When some of the keywords are found, the information is decompressed, and then stored or redirected to specific targets. Therefore, it is useful to have an adaptive compression method with *direct search capabilities*, that is, permitting direct search of the compressed text with a performance close to that achieved by the semistatic techniques.

Our first contribution in this paper is a variant of DETDC, which we call *Dynamic Lightweight ETDC (DLETDC)*. DLETDC has almost the same compression ratio as ETDC and DETDC, but it requires much less processing effort from the receiver than DETDC. As a result, decompression time is now better than that of Ziv-Lempel methods. The key idea is to relieve the receiver of maintaining the model updated as decompression progresses, thus breaking the usual symmetry of statistical adaptive compression. The sender must notify the receiver all the changes on the model. For this to be useful, we design DLETDC so as to maintain its compression ratio while minimizing the required updates to the model.

Our second contribution focuses on the case where the receiver does not need to recover the original text, but just to detect the presence of some keywords in it. We show how DLETDC compressed text can be searched without decompressing it. The search algorithm is even lighter than the decompression algorithm. It needs very little memory and can perform efficient Boyer-Moore-type searching [Boyer and Moore 1977] on the compressed text. We show that searching the compressed text for a set of keywords is much faster (usually twice, and up to 7 times) over the compressed text than over the uncompressed text. This breaks another long-standing assumption that states that only semistatic models permit efficient Boyer-Moore searching on the compressed text. In particular, this is the first adaptive compression scheme that permits searching the compressed text faster than the uncompressed text (and indeed close to searching under semistatic compression).

Our third contribution is the lightweight version of DSCDC, called *Dynamic Lightweight SCDC (DLSCDC)*. As DLETDC, this new technique frees the receiver from the effort of maintaining an updated version of the model. Moreover, being a variant of DSCDC, the sender also deals with all the effort of maintaining parameters $(s, c)$ tuned according to the text that is being compressed [Brisaboa et al. 2008]. The price is that compression time worsens 10% with respect to DLETDC and 5% with respect to DSCDC. However, thanks to its better compression ratio, decompression in DLSCDC is even faster than that of DLETDC, and searches can also be performed very efficiently.

We describe both DLETDC and DLSCDC with sufficient detail to be useful for a practitioner. We also present exhaustive experimental results comparing our methods against the most popular compressors, in terms of compression ratio and compression/decompression speed. Finally, we compare the search speed of DLETDC and DLSCDC against searches in text compressed with semistatic, adaptive, and dictionary-based compressors, as well as in uncompressed text.

Compared to the preliminary version of this work [Brisaboa et al. 2005b], in this

paper we introduce the more complex variant DLSCDC, we give analytical results that explain the performance of our methods, and we present more exhaustive experiments, in particular regarding search times.

Open-source implementations of dynamic lightweight dense code compressors, decompressors, and searchers, are available at `http://rosalia.dc.fi.udc.es/codes/`. As a guide to follow the rest of the paper, Figure 1 presents a dictionary with the acronyms of the different variants of dense codes.

| | | |
|---|---|---|
| Semistatic | ETDC | End Tagged Dense Code |
| | SCDC | $(s, c)$-Dense Code |
| Dynamic | DETDC | Dynamic End Tagged Dense Code |
| | DSCDC | Dynamic $(s, c)$-Dense Code |
| Dynamic lightweight | DLETDC | Dynamic Lightweight End Tagged Dense Code |
| | DLSCDC | Dynamic Lightweight $(s, c)$-Dense Code |
| | DLSCDC$^{Tv}$ | Trivial Dynamic Lightweight $(s, c)$-Dense Code |

Fig. 1. Acronyms of the different variants of dense codes.

## 2. RELATED WORK

We briefly describe the semistatic and dynamic compressors based on dense codes. The reader is referred to [Brisaboa et al. 2007], in the case of the semistatic compressors, and to [Brisaboa et al. 2008], in the case of the dynamic versions, for a complete description, as well as for a complete suite of empirical results.

### 2.1 End Tagged Dense Codes

To compute the codeword of each source word, ETDC uses a semistatic model that is simply the vocabulary (list of source symbols) ordered by frequency. The encoding process is as follows:

—One-byte codewords from 128 to 255 are given to the first 128 words in the vocabulary.

—Words in positions[3] 128 to $128 + 128^2 - 1$ are sequentially assigned two-byte codewords. The first byte of each codeword has a value in the range $[0, 127]$ and the second in range $[128, 255]$.

—Words from $128 + 128^2$ to $128 + 128^2 + 128^3 - 1$ are given tree-byte codewords, etc.

Due to the simplicity of ETDC, fast and simple encode and decode procedures are available. We denote *encode* the function that obtains the codeword $C_i = encode(i)$ for a word at the $i$-th position in the ordered vocabulary; *decode* computes the position $i = decode(C_i)$ in the rank, for a codeword $C_i$. The procedure *encode* requires just 6 lines of source code, whereas *decode* requires 11 lines.

Although its origins are unclear, the coding scheme used in ETDC is not new. It has been used for more than one decade to compress document identifiers in inverted

[3]Note that our word positions start at zero.

indexes. It can be found in the literature as byte-codes (bc) or variable-byte coding (Vbyte) [Williams and Zobel 1999; Culpepper and Moffat 2005].

### 2.2 Dynamic End-Tagged Dense Code (DETDC)

DETDC works as a traditional dynamic compressor. Both sender and receiver maintain the model (in this case, the array of source symbols sorted by frequency). During the compression, if the sender processes a word that is in the $i^{th}$ position of the ordered vocabulary, then it sends $C_i = encode(i)$. Next, the receiver computes $decode(C_i)$ to obtain the position in the ordered vocabulary of the original word. Both ends perform the same process to keep the vocabulary ordered after the occurrence of the last processed word.

When the sender reads a word that is not in the vocabulary yet, it informs the receiver about it by sending a special code (from now on $C_{zeroNode}$) followed by the source word in plain form. Next, such a word is added in the last position of the vocabulary at both ends.

### 2.3 (s,c)-Dense Codes

End-Tagged Dense Code uses 128 target symbols for the bytes that do not end a codeword (*continuers*), and the other 128 target symbols for the last byte of the codeword (*stoppers*). In order to improve the compression ratio, $(s, c)$-Dense Code adapts the number of stoppers and continuers to the word frequency distribution of the text, so that $s$ values are used as stoppers and $c = 256 - s$ values as continuers.

The encoding process is similar to that of ETDC:

—One-byte codewords from 0 to $s - 1$ are assigned to the first $s$ words in the vocabulary.

—Words in positions $s$ to $s + sc - 1$ are sequentially given two-byte codewords. The first byte of each codeword has a value in the range $[s, s + c - 1]$ and the second in range $[0, s - 1]$.

—In general, words from $W_{k-1}^s$ to $W_k^s - 1$ are assigned $k$-byte codewords, where $W_k^s = s\frac{c^k - 1}{c-1}$.

In [Brisaboa et al. 2007; Fariña 2005] a discussion on how to obtain the $s$ and $c$ values that minimize the size of the compressed text for a specific word frequency distribution can be found.

The coding and decoding algorithms are the same as those of ETDC, changing the 128 values of stoppers and continuers by $s$ and $c$ where appropriate. Thus on-the-fly *encode* and *decode* algorithms are also available.

### 2.4 Dynamic (s,c)-Dense Codes (DSCDC)

The dynamic version of SCDC works just like DETDC except that at each step of the compression/decompression processes, DSCDC has to check whether the current value of $s$ (and $c$) remains well tuned or if it should change. To keep $s$ and $c$ tuned as compression/decompression progresses, DSCDC compares the size of the compressed text assuming that the values $s - 1$, $s$, and $s + 1$ were used for coding up to current word $w_i$. If the compressed text becomes smaller by using either $s - 1$ or $s + 1$ instead of $s$, then $s$ is changed properly from now on. Therefore, during

the coding/decoding of a word/codeword, the value of $s$ changes at most by one. Other heuristics are described in [Fariña 2005].

## 3. DYNAMIC LIGHTWEIGHT END TAGGED DENSE CODE

*Dynamic Lightweight End Tagged Dense Code* (DLETDC) is based on DETDC, but it avoids the overhead of keeping the model up-to-date in the side of the receiver. This makes it extremely convenient in scenarios where the bandwidth is low and the receiver has little processing power, such as in mobile applications. The price to pay is a very slight increase in the processing cost of the sender and a negligible loss of compression ratio.

In DLETDC, only the sender keeps track of the frequency of each symbol and maintains the vocabulary sorted by frequency. The receiver, instead, only stores an array of words indexed by their codewords, with no frequency information. When a codeword arrives, the receiver decodes it using the standard *decode* procedure and obtains the corresponding word position. The receiver does not update the model at all. Therefore, the sender should inform the receiver of any change in the words↔codewords mapping. Note that changes in the codeword assignments upon frequency changes are necessary to maintain good compression ratios. However, the number of exchanges in the vocabulary is large enough to affect the compression ratio if all of them have to be informed to the receiver, where they also require some effort to be processed. Hence, we seek at minimizing the number of exchanges without affecting the compression ratio.

Our basic idea is to carry out the changes in the words↔codewords mapping only when the increment in the frequency of a word $w_i$ makes it necessary to encode it with a codeword shorter than its current codeword $C_i$. There is no correspondence between the word rank (position in the vocabulary sorted by frequency of the sender) and its codeword anymore, because words may vary their positions without changing their codewords. That is, changes in the rank of a word do not produce changes in its codeword except when the codeword must be shorter. Thus, the sender must maintain an explicit words↔codewords mapping, to encode the words.

When the codeword $C_i$, corresponding to the word $w_i$, must be shortened, we look for the last word $w_j$ such that $|C_j| < |C_i|$, and swap $C_i$ and $C_j$. Thus, DLETDC needs two special codewords, $C_{zeroNode}$ and $C_{swap}$. $C_{zeroNode}$ works as in DETDC. $C_{swap}$ specifies that the receiver should swap the words at the positions given by the two codewords that follow $C_{swap}$. That is, $\langle C_{swap}, C_i, C_j \rangle$ indicates that from now on $w_i$ is represented by $C_j$, and $w_j$ by $C_i$. This is done in the receiver by a simple swap of words at positions $i$ and $j$ of the vocabulary array.

Codewords $C_{zeroNode}$ and $C_{swap}$ can be any unused values, for example the first two unused codewords. Another choice is to give them fixed values during the whole process. In our real implementation, we used the last two codewords of 3 bytes, since 3 bytes are by far more than enough in all our experimental corpora. This choice favors the search speed, as we will see later, although we lose some compression, because if the next free codewords were used, the length would be 1, 2, or 3. Yet this has few consequences: using the Calgary corpus (described in Section 5), which has a size of 2 Mbytes, the *3-byte version* obtains a file of 1,068,937 bytes (50.16% of compression ratio), whereas the *first free codewords*

*version* produces a file of 1,051,868 bytes (49.35%). In a larger file (ZIFF) of 176 Mbytes, the effect is hardly noticeable, obtaining 62,754,382 bytes (33.88%) and 62,737,208 bytes (33.87%) respectively.

EXAMPLE 3.1. Figure 2 shows an example of the process carried out by the sender. Assume that, after compressing some text, word `"step"` is at position 127 in the sorted vocabulary, `"many"` is at 128, and `"bit"` at 129, all of them with frequency 19. The text to be compressed next is `"by step bit by bit"`.

After reading `"by"`, we check that such a word is not in the vocabulary, thus we add it at the last position (130) and give it the codeword $C_{130}$. We inform the receiver of this addition by sending $C_{zeroNode}$ and word `"by"` in plain form. The next word (`"step"`) is already in the vocabulary and increases its frequency by 1. Next, we reorder the vocabulary. Assume that `"step"` remains at position 127. Then we send codeword $C_{127}$.

The first occurrence of `"bit"` increases its frequency to 20 and then, after the reorganization, `"bit"` is relocated at position 128 by swapping it with `"many"`. However, since $C_{128}$ (the codeword representing `"many"`) and $C_{129}$ (the code for `"bit"`) have the same size (2 bytes), we continue using $C_{128}$ for `"many"` and $C_{129}$ for `"bit"`. Therefore, $C_{129}$ is sent. For the next word (`"by"`) we send $C_{130}$.

The next occurrence of `"bit"` places it at position 127, whose codeword has 1 byte. Then, since in this case $C_{127}$ and $C_{129}$ have different sizes, we swap those codewords, associating `"step"` with $C_{129}$ and `"bit"` with $C_{127}$. To inform the receiver of this change, we send the tuple $\langle C_{swap}, C_{129}, C_{127}\rangle$. $C_{swap}$ warns the receiver to expect two codewords that should be exchanged. The receiver also understands that the codeword after the $C_{swap}$ indicates the word actually read.

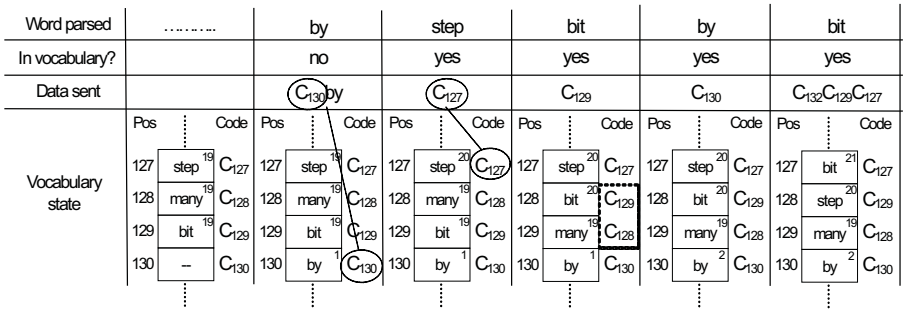



Fig. 2. Transmission of `"... by step bit by bit"`.

We can find some similar ideas in a previous adaptive coder [Turpin and Moffat 1997a]. In order to improve the throughput of the process, the code only changes when the "approximate frequency" of a source symbol varies after a new occurrence. As DLETDC, which avoids changing the codeword assignment unless compression ratio would suffer, the use of this approximate frequency results in fewer codeword changes than in the case of using the actual frequency. This approximate coding assumes a controlled loss in compression ratio due to this delayed update of codewords. Yet, DLETDC does not lose compression due to its deferred codeword assignments, as it assigns shorter codewords to source symbols as soon as the ETDC compression scheme can take advantage of such a new codeword assignment.

### 3.1 Data structures

The main requirement to design the data structures for the sender is the need to identify *blocks* of words with the same frequency, and to be able to promote a word to the next block fastly when its frequency increases.

The data structures used by the sender and their functionality are shown in Figure 3. The hash table permits fast searching for any source word $w_i$ to obtain its current position $i$ in the rank (ordered vocabulary), as well as its current frequency and codeword. It keeps in *word* the source word, in *posInVoc* the position of the word in the rank, in *freq* its frequency, and in *codeword* its codeword.

*posInHT* is a vector where the $i^{th}$ entry points to the position in the hash table that stores the $i^{th}$ most frequent word in the vocabulary. Finally, vector *top* keeps a pointer to the entry in *posInHT* that points to the first (top) word with each frequency. If there are no words of frequency $f_i$ then $top[f_i]$ points to the position of the first word $j$ in the ordered vocabulary such that $f_j < f_i$. A variable *last* storing the first unused vocabulary position is also needed.

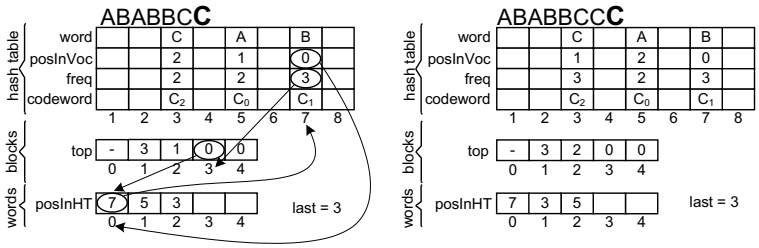



Fig. 3. Data structures of the sender.

When the sender reads a word $w_i$, it uses the hash function to obtain its position $p$ in the hash table. If $w_i$ is a new word, the algorithm sets $word[p] = w_i$, $freq[p] = 1$, $posInVoc[p] = last$, $codeword[p] = C_{last}$ ($C_{last} = encode(last)$), and $posInHT[last] = p$. Then, the variable *last* is increased and finally the codeword $C_{zeroNode}$ is sent followed by the word in plain form.

If $w_i$ is already in the hash table, $word[p] = w_i$. After reading $f = freq[p]$, $freq[p]$ is incremented. The position of $w_i$ in the vocabulary array is obtained as $i = posInVoc[p]$ and its codeword as $C_i = codeword[p]$. Now, word $w_i$ must be promoted to the next block (the one containing words with frequency $f + 1$). For this sake, the sender algorithm finds the word that is the head of its block $j = top[f]$, and the corresponding position $h$ of such a word in the hash table $h = posInHT[j]$. Now, it is necessary to swap words $i$ and $j$ in vector $posInHT$. This requires exchanging $posInHT[j] = h$ with $posInHT[i] = p$, setting $posInVoc[p] = j$ and $posInVoc[h] = i$. Once the swapping is done, $j$ (the new position in the ordered vocabulary of $w_i$) is promoted to the next block by setting $top[f] = j + 1$.

If the codeword $C_j$ has the same length as $C_i$, then $C_i$ is sent because it remains as the codeword of $w_i$, but if $C_j$ is shorter than $C_i$, then $codeword[h] = C_j$ and $codeword[p] = C_i$ are swapped and the sequence $\langle C_{swap}, C_i, C_j \rangle$ is sent. The receiver will understand that words at positions $i$ and $j$ in its vocabulary array must be swapped and that word $w_i$, which from now on will be encoded as $C_j$, was sent.

In the receiver, a simple words array *word* and a variable *last* are the only structures needed, as explained. Words in *word* array are sorted by codeword, and two words are swapped always following sender instructions. New words are introduced in the vocabulary array as they arrive, always at the last position. Therefore, there is an implicit mapping between word position and codeword, as in ETDC. This fact permits using the same *decode* procedure of ETDC and DETDC.

The pseudocodes for both sender and receiver algorithms are available at the web address `http://rosalia.dc.fi.udc.es/codes/lightweight.html`.

### 3.2 Searching the compressed text

The problem of performing direct search over text compressed with previous dynamic methods is that the codeword used to encode a specific word changes many times along the process. Therefore, following those changes requires an effort close to that of just decompressing the text and then searching it.

However, in DLETDC we expect very few codeword swaps, thus codewords assigned to words should vary much less frequently than in previous adaptive techniques. Moreover, those swaps are explicitly marked with a $C_{swap}$. This makes it possible to scan the arriving text looking for some specific patterns, paying the overhead of re-preprocessing the patterns upon such changes. The first occurrence of a searched word will appear in plain form, preceded by codeword $C_{zeroNode}$. At that point, codeword $C_{last}$ becomes the pattern we must look for to find our word (and *last* is the number of $C_{zeroNode}$ codewords we have seen at that point). That codeword may change again later, but such a change will be signaled by the codeword $C_{swap}$. Thus, the scanning algorithm can easily follow the evolution of the search patterns across the compressed text.

We apply a Boyer-Moore-type search algorithm, specifically *Set Horspool* [Horspool 1980; Navarro and Raffinot 2002]. We have to use the multi-pattern version of *Horspool*, since we always have to search for $C_{swap}$ (plus the searched patterns) in order to detect codeword changes. *Set Horspool* is the best choice to search for a moderate number of short patterns on a large alphabet (in our case, it is close to uniformly distributed over 256 values). *Set Horspool* builds a reverse trie with the search patterns to speed up comparisons against the text.

However, we have to consider several special issues when searching DLETDC compressed text. Assume that we are searching for patterns $p_1, p_2, \ldots, p_k$. We use $P(p_i)$ to denote codeword $C_{zeroNode}$ followed by the plain version of the pattern, and $C(p_i)$ to denote the codeword representing $p_i$ at a certain moment. Note that $P(p_i)$ cannot be confused with a codeword because it starts with $C_{zeroNode}$. Actually, we do not search for the full $P(p_i)$ strings, but just by $C_{zeroNode}$. The reason is that we always have to look for $C_{zeroNode}$ to keep track of the current vocabulary size (value *last*, see next paragraph). In any multi-pattern Boyer-Moore-type searching, when longer and shorter patterns are sought, the most efficient choice is to truncate all to the shortest length and verify them upon occurrence of their truncated version. The truncated part of the $P(p_i)$ is $C_{zeroNode}$ (choosing other truncated substring would only increase the probability of verifying). We store all the search patterns $p_i$ that have not yet been found in an auxiliary trie, which is used to verify the occurrence of those patterns upon each occurrence of $C_{zeroNode}$.

Initially, the search trie has only the codewords $C_{zeroNode}$ and $C_{swap}$. Each time

$C_{zeroNode}$ codeword is recognized, the algorithm traverses the auxiliary trie with the characters of the plain string that follows $C_{zeroNode}$. If some $p_i$ is detected, then we have found the first occurrence of $P(p_i)$ in the text. The search trie is updated by inserting the corresponding pattern $C(p_i)$ (which initially is the current $C_{last}$ value), and the auxiliary trie is also updated to remove the pattern $p_i$. In any case, we restart the search at the end of the plain string (recognized by a zero-character terminator). Note that, when the auxiliary trie becomes empty, this check is not needed anymore. At this point we can remove $C_{zeroNode}$ from the search trie, as we do not seek for any new pattern nor need to know $C_{last}$ again.

The codeword $C_{swap}$ is always in the search trie, as the codeword of a search pattern can be changed by the sender by using $C_{swap}$ as the escape codeword. Upon finding a $C_{swap}$, the search algorithm reads the next two codewords and checks if one (or both) of them is in the trie. If it is, the trie is updated to replace $C(p_i)$, the current codeword of the searched pattern $p_i$, by its new codeword.

As mentioned, we preferred to use for $C_{zeroNode}$ and $C_{swap}$ the last 3-byte codewords because it leads to a better search speed. The improvement in search speed comes from two combined effects. First, both $C_{zeroNode}$ and $C_{swap}$ must be explicitly represented in the trie, so if we used the first two free codewords to represent them, we would have to update the search trie each time a new word arrived. Second, using fixed 3-byte codewords permits longer shifts in the Boyer-Moore-type algorithm.

### 3.3 Swaps and evolution of new words

One key issue to understand the compression performance of DLETDC is to determine the overhead it produces over DETDC due to the need to notify the swaps. In this section we show, both experimentally and analytically, that this overhead is very low and vanishes as more and more text is transmitted.

The analytical result is based on Zipf's law [Zipf 1949], an empirical law widely accepted in IR, that roughly approximates the distribution of words in the vocabulary. According to Zipf's law, the $i$-th most frequent word in a text of $n$ words appears $An/i^\theta$ times, where $\theta > 1$ is a constant that depends on the text size, and $A = \frac{1}{\sum_{i\geq 1} 1/i^\theta}$ is a correction factor to ensure that the frequencies add up to $n$.

We recall that, using a SCDC or ETDC coding scheme, there are some positions in the sorted vocabulary, $W_k^s = s\frac{c^k-1}{c-1}$ ($s = c = 128$ in ETDC) for $k = 1,\ 2, \ldots$, where the codeword size changes. To be aligned with Zipf's formulas, let us note in this section the word positions as starting in 1, not 0. Thus, the word at position $W_k^s$ is encoded with a codeword of length $k$, whereas the word at position $W_k^s + 1$ is encoded with $k + 1$ bytes.

In DLETDC, if a change in frequency makes a symbol move from a position after $W_k^s$ to a position before (or equal to) $W_k^s$, then a $C_{swap}$ symbol will be transmitted. In other words, this swap will occur if the incremented frequency of a symbol at a position $j > W_k^s$ is not smaller than the frequency of the symbol at position $i = W_k^s$. Assuming Zipf's law, this is equivalent to the condition $An/j^\theta + 1 \geq An/i^\theta$. Solving for $j$, the symbols that will produce a swap around codeword $i = W_k^s$ when incremented are those at positions $i + 1$ to $j = i(An)^{1/\theta}/(An - i^\theta)^{1/\theta}$. The probability of each new symbol falling between $i + 1$ and $j$ is

$$A \sum_{x=i+1}^{j} 1/x^\theta \leq A \int_{x=i}^{j} 1/x^\theta dx = \frac{A}{\theta-1}\left(\frac{1}{i^{\theta-1}} - \frac{1}{j^{\theta-1}}\right)$$
$$= \frac{A}{(\theta-1)i^{\theta-1}}\left(1 - \left(\frac{An - i^\theta}{An}\right)^{1-1/\theta}\right),$$

where $n$ is the current size of the text. Therefore, the total number of swaps due to crossing position $i$ as the text size grows from 1 to $n$ is at most

$$\frac{A}{(\theta-1)i^{\theta-1}} \int_{x=i^\theta/A}^{n} \left(1 - \left(1 - \frac{i^\theta}{Ax}\right)^{1-1/\theta}\right) dx, \quad (1)$$

where we have disregarded the first $i^\theta/A$ text words, necessary to reach the vocabulary cell $i$ under Zipf's law. The integral could be solved only for some particular values of $\theta$. In particular, for $\theta = 2$, the primitive is

$$\left(x - \sqrt{x\left(x - \frac{i^2}{A}\right)} + \frac{i^2}{2A}\ln\left(\sqrt{A}\left(x - \frac{i^2}{2A} + \sqrt{x\left(x - \frac{i^2}{A}\right)}\right)\right)\right) = \frac{i^2}{2A}\ln x + O(1)$$

(note we are treating $A$ and $i$ as constants and $x$ as the asymptotic variable). Replacing in Eq. (1), we get that for $\theta = 2$ the total number of swaps is of the form $\frac{i}{2}\ln n + O(1)$. The integral can also be solved for $\theta = 1/k$ and $\theta = 2/k$, for any integer $k \geq 1$. In all these cases, Eq. (1) gives $\frac{i}{\theta}\ln n + O(1) = \frac{W_k^s \ln n}{\theta} + O(1)$. We conjecture this is the general solution, and verified the accurateness of our conjecture for a large number of values of $\theta > 1$, by integrating numerically.[4].

**Conjecture:** Under Zipf's law with parameter $\theta$, the number of swaps produced between codewords of length $k$ and $k+1$ in a Dynamic Lightweight Dense Code with parameters $(s, c)$, after processing $n$ words, is $\frac{W_k^s}{\theta}\ln n + O(1)$.

Thus, we have good reasons to expect that the number of swaps due to each $W_k^s$ is $\Theta(\log n)$ and also proportional to $W_k^s \approx 128^k$. [5]

We have verified this logarithmic growth experimentally with the ZIFF collection, described in Section 5. This file has 40,627,132 words, and a vocabulary of 237,622 (different) words. The $\theta$ value that better approximates Zipf's law on this collection is $\theta = 1.744$. With this vocabulary size, and using $s = c = 128$, there are only two limits between zones, at $W_1^s = 128$ and $W_2^s = 128 + 128^2$.

In the process, 31,772 swaps were produced. This means that only 0.078% of the frequency changes implied a swap. In addition, most of these swaps were produced in the first stages of the compression, as it can be seen in Figure 4. As a comparison, in the case of DETDC, the number of swaps was 8,777,606 (21.605%

[4]If we opt for a simplified lower bound given by multiplying by $n$ instead of integrating over all the $n$ values, we formally obtain the form $i/\theta$.

[5]The original Zipf law used $\theta = 1$, which has been shown not to hold in natural language text [Baeza-Yates and Ribeiro-Neto 1999]. Yet, despite the development is rather different, the result for $\theta = 1$ can be proven to be $i \ln n + O(1)$, thus fitting smoothly in our general conjectured formula.

of the 40,627,132 processed words). That is, DETDC produced around 278 times more codeword changes than DLETDC.

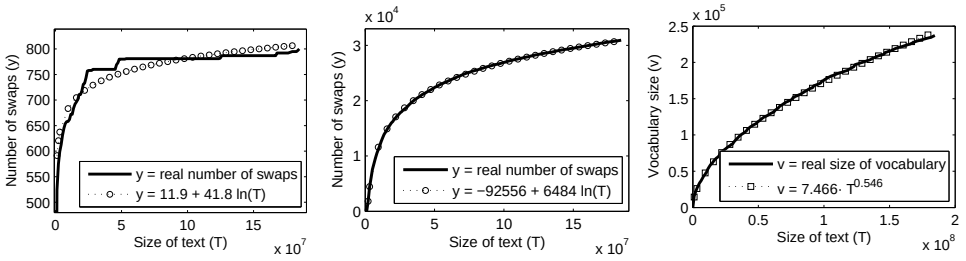



Fig. 4. Swaps of (a) 1 and 2 byte, and (b) 2 and 3 byte codewords. (c) Number of new words.

Note that most of the changes (30,971) are between codewords of size 2 and 3. This is expected from our formula: The shape of the Zipf distribution predicts a bigger difference in frequency between words $128^{th}$ and $129^{th}$ than between words in positions $128 + 128^2$ and $128 + 128^2 + 1$.

Figure 4(a) shows the number of swaps between codewords of length 1 and 2. The curve can be reasonably approximated with the model $y = b + a \cdot \ln T$ for $a = 41.8$ and $b = 11.9$, where $T$ is the text size in bytes[6].

Figure 4(b), which plots the number of swaps between codewords of length 2 and 3, exhibits an almost perfect logarithmic growth of the form $y = b + a \cdot \ln T$, with an excellent fitting for $a = 6,484$ and $b = -92,556$.

The values predicted with the formula $a = \frac{W_k^s}{\theta} \ln n$ are 73.4 for Figure 4(a) and 9,468 for Figure 4(b), which are of the same order of magnitude of the actual numbers. This is remarkable for a rather coarse law such as Zipf's, and confirms our theoretical predictions.

Figure 4(c) also shows that the number of different words ($v$), as expected, follows Heaps' law [Heaps 1978], $v = K \cdot T^{\beta}$, $0 < \beta < 1$, for $K = 7.468$ and $\beta = 0.546$.

Note that we have used Heaps' and Zipf's empirical laws to carry out our analysis. These have been successfully used for decades to model the behavior of Natural Language text collections. Of course, one can build worst-case (artificial) texts where these laws do not hold. More interesting is the fact that there could be Natural Language text collections that deviate from those laws, for example those obtained by concatenating texts from different sources or languages and then reordering the documents, such that the word statistics vary relatively fast along the text.

We empirically studied the behavior of DLETDC in these scenarios. Our first experiment entailed reordering the documents of collection ALL (described in Section 5), which includes English articles from widely different areas. We made two partitions: The first divided the corpus into documents of around 1 Kbyte, and the second of around 10 Kbytes. Those documents were then randomly reordered. The compression ratio achieved by DLETDC was 33.69% without reordering, 33.70%

[6]Note that $n$, the total number of words, depends on $T$, the text size ($T \approx 4.5n$).

when reordering 1 Kbyte partitions, and 33.68% when reordering 10 Kbyte partitions. The second experiment used a multilingual corpus obtained by concatenating collection ALL with a corpus of news from the Spanish agency EFE. DLETDC compressed the resulting corpus of 1.5 Gbytes to 35.63%. When reordering 10 Kbyte partitions this raised to 35.64%, and to 35.67% with 1 Kbyte partitions.

This shows that text collections should be rather particular to make a significant difference in this respect. Note that, in monolingual corpora, changes in short codewords (1 vs 2 bytes) are very unlikely, as these depend on the frequency of the stopwords, which is independent on the topic of the documents. In the multilingual experiment, the compression ratio worsens mainly due to the increase of the vocabulary size, revealing that frequency changes do not have a significant impact.

## 4. DYNAMIC LIGHTWEIGHT (S,C)-DENSE CODES

Applying the same guidelines followed to create DLETDC from DETDC, a *Dynamic Lightweight (s,c)-Dense Code* (DLSCDC) can also be developed.

We developed a first *trivial* approach, which we denote DLSCDC$^{Tv}$, to improve the compression ratio achieved by DLETDC, based on the experiences of applying SCDC and DSCDC to texts, which showed that the optimal value of the parameter $s$ usually lies in the range $s \in (180 \ldots 200)$. DLSCDC$^{Tv}$ simply fixes $s = 190$ (and $c = 66$), and applies the encoding/decoding scheme of SCDC instead of that of ETDC. Results are interesting, because the compression ratio of DLETDC is improved, whereas the compression and decompression speed worsen just very slightly.

A true DLSCDC, however, should be able to dynamically adapt the parameters $s$ and $c$ to any source text. Therefore, apart from the process carried out by DLETDC, DLSCDC must also deal with maintaining $s$ (and $c$) well-tuned. Obviously, in order to obtain a lightweight receiver, the sender must deal with that effort, and notify the receiver of the changes. A new escape codeword, $C_{sChange}$, is added to warn the receiver about a change on $s$ (and $c$). As in DLETDC, codewords $C_{zeroNode}$, $C_{swap}$, and $C_{sChange}$ can be any unused codewords. In our implementation, we used fixed 4-byte codewords. For usual values of $s$ around 190, using these escape codewords allows to encode more than 50 million words[7]. Again, this choice has little effects in the compression ratio. The compressed version of the Calgary corpus has 1,105,005 bytes (51.82% of compression ratio) with *4-byte* escape codewords, 1,064,346 bytes (49,94%) in the case of using *3-byte* escape codewords, and 1,047,092 bytes (49,13%) with the *first free codewords version*. As expected, the effect is even less noticeable in a larger corpus like ZIFF, where the compressed version occupies: 61,998,933 bytes (33.47%), 61,711,832 bytes (33.31%), and 61,694,413 bytes (33.30%), respectively, depending on the escape codewords used.

As in DLETDC, the vocabulary of the sender is maintained ordered by codeword length. An important difference with respect to DLETDC is that the limits of the ranges of words that are encoded with the same number of bytes ($k$) are not fixed

[7]In DLETDC, $C_{zeroNode}$ and $C_{swap}$ were 3-byte codewords. This allows to code up to $W_3 = 2,113,664$ different words, enough even for huge texts. Yet, by setting, for example, $s$ to 190 (and $c = 66$), the amount of words that can be coded with up to 3-byte codewords decreases drastically to $W_3^{190} = 840,370$ words. Therefore, in DLSCDC, we use 4-byte escape codewords.

in DLSCDC. That is, the range of word positions that are encoded with $k$ bytes, $[W_{k-1}^s, W_k^s)$, depends on $s$ and $c$. Note that, indeed, all the codes depend on a single change in $(s, c)$; we will return to this issue later. In particular, when $s$ changes from $s_0$ to $s_1$, words in the range $[W_k^{s_0}, W_k^{s_1})$, $k > 0$ and $W_k^{s_0} < W_k^{s_1}$, (or in $[W_k^{s_1}, W_k^{s_0})$, if $W_k^{s_1} < W_k^{s_0}$) might be left with a codeword of inappropriate length; obviously, this worsens compression. The following example illustrates.

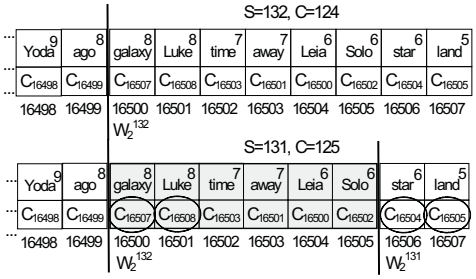



Fig. 5. Scenario after a change of $s$ and $c$ in DLSCDC.

Example 4.1. Figure 5 shows the state of the vocabulary of the sender at a given point of the compression process. For each word, we show its number of occurrences, the word itself, the codeword that is currently assigned to it, and its position in the ordered vocabulary. For example, the word "`Yoda`" is located at position 16,498, has appeared 9 times, and its current codeword is $C_{16498}$. Having $s = 132$, words from position 16,500 onwards are encoded with 3-byte codewords, and their codewords will not change unless new occurrences make them move before position 16,500 ($W_2^{132}$). This works fine if $s$ does not change. However, if $s$ decreases from 132 to 131, words in the range $[16500, 16505]$ (i.e., $[W_2^{132}, W_2^{131})$) might have codewords of an inappropriate length. Specifically, words "`galaxy`" and "`Luke`" now deserve 2-byte codewords, since they are ranked before $W_2^{131}$, yet they still have 3-byte codewords, since $C_{16507}$ and $C_{16508}$ correspond to positions after $W_2^{131}$.

Due to the previous misplacement, there should be words having the opposite situation. That is, words "`star`" and "`land`" are encoded with 2-byte codewords, whereas they deserve 3-byte codewords. Observe that after the change on $s$, their codewords ($C_{16504}$ and $C_{16505}$) correspond now to positions prior to $W_2^{131}$, that is 2-byte codewords. However, their positions are ranked either at or after $W_2^{131}$, which implies that they should have 3-byte codewords.

Therefore, our initial idea was that, after each change on the parameter $s$, the sender should swap the codewords of all those words that are assigned wrong-length codewords, and then notify which words need to be swapped in order to keep the receiver synchronized with the sender. In practice, we decided to perform swaps only for the words with a 1-byte codeword that deserve a 2-byte codeword and those that own a 2-byte codeword but are ranked before $W_1^s$. We avoid swapping words around $W_k^s$, $k > 1$, because the number of swaps that could be needed among words around $W_2^s$, and also around $W_3^s$, is usually too high (for example, assuming that at a certain moment the encoder is using codewords with up to 4 bytes, increasing $s$ from 190 to 191 could produce up to $1 + 122 + 20,903 = 21,026$

swaps). Moreover, since the number of occurrences of words upon $W_3^s$ is very low (usually $1-2$) the number of bytes needed to notify all those swaps would be larger than the extra bytes paid for encoding one occurrence of some of these low-probability words with one extra byte. In addition, finding the words with a wrong-length codeword and performing the swaps implies a cost in time that could lead to non-real time transmission.

In order to empirically verify the convenience of this heuristics, we used again the ZIFF collection. This file has 185,220,215 bytes, and during its compression, the DLSCDC compressor performs 3,519 changes on $s$. Running DLSCDC performing swaps only between words with codewords of 1-2 bytes, yields a compressed file of 61,988,933 bytes (33,47% of compression ratio). When we force our compressor to swap codewords of all sizes, the compressed file has 62,513,913 bytes (33,75%). That is, the cost to notify the swaps outweighs the savings due to better coding.

Observe that, after a change on $s$ and considering only the misplacement of codewords of 1-2 bytes, the interval where there can be words with codewords of wrong length ($[W_1^{s_0}, W_1^{s_1})$ or $[W_1^{s_1}, W_1^{s_0})$) only includes one position, since $s$ changes at most by one. Therefore, changing $s$ from $s_0$ to $s_1$ will imply notifying only one swap to the receiver. This swap will involve the words at positions $\beta$ and $posSwap$, where $\beta = min(s_0, s_1)$, and $posSwap$ is the rank of the word that is currently encoded as $C_\beta$. Next, the sequence $\langle C_{sChange}, s_1, C_{posSwap}\rangle$ is transmitted. Then, the receiver performs a swap between the words at positions $\beta$ and $posSwap = decode(C_{posSwap})$. The encoder sends $C_{posSwap}$ (instead of $posSwap$), as it could require more than one byte to be encoded.

### 4.1 Data structures

The sender in DLSCDC uses basically the same data structures shown in DLETDC. However, vector *codeword*, which stored a codeword $C_x$ to handle the explicit mapping word↔codeword in DLETDC, keeps just the number $x$ in DLSCDC. Using this number $x$, the corresponding codeword can be obtained as $C_x = encode(x, s)$. Note that this is mandatory in DLSCDC as the parameter $s$ can change throughout the compression process. Moreover, a new vector *cwPos*, containing $v$ elements, permits locating the word $i$ in the vocabulary that is currently being encoded as $C_j$. That is, $cwPos[j] = i$ iff $codeword[posInTH[i]] = j$. This allows us to locate the word $posSwap = cwPos[\beta]$ in $O(1)$ time, which is needed each time the parameter $s$ changes. The data structures used by the sender are shown in Figure 6.

The sender process in DLSCDC is similar to that of DLETDC, with two main differences: *i)* It has to maintain vector *cwPos* updated to follow the changes in the position of words and codewords, and *ii)* the sender has also to modify the value $s$ (and notify it to the receiver) when needed.

When the sender reads a word $w_i$, it hashes it to $p = hash(w_i)$. Then if $w_i$ is a new word, the only difference with respect to DLETDC process for this case is that the *cwPos* entry should be initialized, that is $cwPos[last] = last$.

If $w_i$ was already in the vocabulary, the sender obtains its rank as $i = posInVoc[p]$ and its codeword as $C_i = encode(codeword[p], s)$. Promoting $w_i$ to the next frequency group involves the same operations as in DLETDC. This process includes obtaining $f = freq[p]$, the rank of the first word with frequency $f$ ($j = top[f]$), the position in the hash table of such a word ($h = posInHT[j]$) and its codeword

($C_j = encode(codeword[h], s)$). If $C_j$ has the same length as $C_i$, then $C_i$ is sent and $cwPos[codeword[p]]$ is swapped with $cwPos[codeword[h]]$ to maintain vector $cwPos$ up-to-date. However, if $C_j$ is shorter than $C_i$ then $codeword[h] = C_j$ and $codeword[p] = C_i$ are swapped and the $C_{swap}$ tuple is sent.

In Figure 6, assuming that the text "ABBCDEF" has already been processed (left), it is shown how the sender deals with a new occurrence of word "D". Dark boxes indicate data that is modified during the process. Since "D" is located at slot 4 in the hash table, we have $p = 4$, $i = 3 = posInVoc[4]$, $f = 1 = freq[p]$, and $j = top[1] = 1$. Then we obtain $h = 5 = posInHT[j]$, and compute $C_i = C_3 = encode(codeword[4], s)$ and $C_j = C_0 = encode(codeword[5], s)$. Next, we swap $posInHT[1] \leftrightarrow posInHT[3]$ and $posInVoc[4] \leftrightarrow posInVoc[5]$, and promote word "D" to the group of frequency 2 by setting $top[1] = 2$ and $freq[4] = 2$.

Now the sizes of $C_i$ and $C_j$ have to be compared to check if a swap has to be performed. Assuming that in that moment $s = 4$ (Figure 6 (b)), $C_i$ and $C_j$ have the same size. Hence no swap between codewords takes place, and only an exchange between $cwPos[codeword[4]]$ and $cwPos[codeword[5]]$ is done. However, if we assume that $s = 3$ (Figure 6 (c)), then the size of $C_j$ is 1, whereas the size of $C_i$ is 2. Therefore, a swap between $codeword[4]$ and $codeword[5]$ is performed.

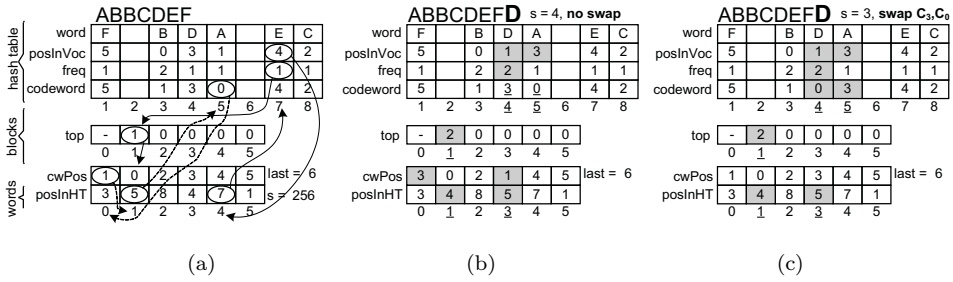



Fig. 6. Data structures of the sender.

After having processed the word $w_i$, the sender calls the algorithm that modifies the value of $s$ if needed. When the value of $s$ changes, the sender has to swap the codewords associated to the words ranked $\beta$ and $posSwap = cwPos[\beta]$, and notify such a swap and the new value of $s$ to the receiver. As explained in the previous section, if the value of $s$ was decreased, then $\beta = s$ is set, otherwise $\beta$ is set to the previous value of $s$. Once $\beta$ is known, we find $posSwap = cwPos[\beta]$. Then the slots in the hash table associated to $\beta$ and $posSwap$ are obtained as $p = posInHT[posSwap]$ and $h = posInHT[\beta]$ respectively. Next the codewords associated to $\beta$ and $posSwap$ are swapped by exchanging $codeword[p]$ with $codeword[h]$, and vector $cwPos$ is updated by swapping $cwPos[\beta]$ with $cwPos[codeword[h]]$. Finally, the receiver is informed that the words in positions $\beta$ and $codeword[h]$ have to be exchanged, and is also notified about the new value of $s$. As seen, this is done by sending $\langle C_{sChange}, s, C_{posSwap}\rangle$. Figure 7 shows the data structures of the sender that are modified when $s$ is decreased from 209 to 208. Again changing data is shown darkened.

Apart from the variable $s$, the data structures used by the receiver are exactly the same as in DLETDC. Therefore, only a *word* vector, and the variables *last* and

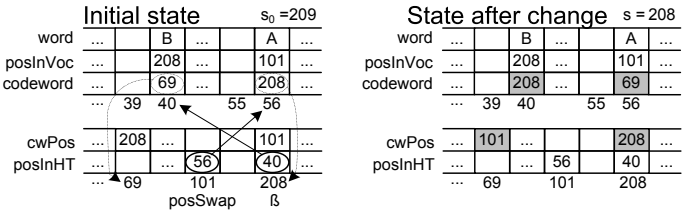



Fig. 7. Data structures of the sender modified after a change on $s$.

$s$ are needed. As in DLETDC, the receiver has only to decode each codeword $C_i$ to obtain the rank $i = decode(C_i, s)$ of the word $w_i$ associated to $C_i$, and to follow sender instructions when an escape symbol is received.

The pseudocodes for both sender and receiver algorithms are available at the web address `http://rosalia.dc.fi.udc.es/codes/lightweight.html`.

*Removing the cwPos vector.* Based on our experimental results, which showed that the number of changes of the parameter $s$ is very small, an improvement of the DLSCDC compressor can be done. As shown, maintaining *cwPos* up-to-date upon swaps between words or codewords, involves performing some operations each time a word is processed. This additional task leads to some loss in the compression time of DLSCDC with respect to DLETDC.

In practice, using a sequential search for the value *posSwap* is much simpler and faster than maintaining the vector *cwPos*. Note that this sequential search does not need to start at the beginning of the vocabulary and traverse all of it, but can be done in a smarter way. On the one hand, if $s$ decreases from $s_0$ to $s_0 - 1$ then $\beta = s_0 - 1$, and it is known that $posSwap \leq \beta$. Therefore, we have only to search for *posSwap* in the range $[0 \ldots \beta]$ (in decreasing order). On the other hand, when $s$ is increased from $s_0$ to $s_0 + 1$, we know that $\beta = s_0$ and $posSwap \geq \beta$. In such a case, the search process is performed through the range $[\beta \ldots v)$. Such a search process involves only iterating over the words $i$ in the chosen range while $codeword[posInHT[i]] \neq \beta$. At the end, we obtain $posSwap = i$.

In practice, searching for *posSwap* sequentially instead of using vector *cwPos* improves the compression speed by around 6–8%. Thus, the empirical results for DLSCDC in Section 5 refer to the faster and simpler version of the compressor.

### 4.2 Searching text compressed with DLSCDC

Searches using Boyer-Moore type algorithms can also be performed directly inside the text compressed with DLSCDC. However, there exists one important difference with respect to the searches performed over text compressed with DLETDC.

Apart from the detection of $C_{zeroNode}$, $C_{swap}$, and the codewords $C_i$ associated to any search pattern $p_i$, when using DLSCDC the searcher must detect the codeword $C_{sChange}$, which indicates that a change on the parameter $s$ takes place. Note that when $s$ changes from $s_0$ to $s_1$, the $(s, c)$-Dense encoding varies for all the symbols. Therefore, the searcher has to remove from the trie all the codewords $C_i$ associated to the search patterns $p_i$ that have already appeared in the text, and add the updated codewords $C_i^{s_1} = encode(decode(C_i, s_0), s_1)$. Of course, since any $C_{sChange}$ implies also a swap between two codewords $C_\beta$ and $C_{posSwap}$, the searcher

swaps the codewords associated to any search pattern $p_i$ such that $C_i = C_{posSwap}$ or $C_i = \beta$.

Another peculiarity of the DLSCDC code defined up to now is that it is not yet searchable (using a Boyer-Moore type algorithm), as false matchings might occur. However, they are easily avoided by ending each sequence $\langle C_{sChange}, s, C_{posSwap} \rangle$ with a byte set to *zero* (to ensure that it is always a stopper value). The problem could appear because, when $s$ decreases, the last byte of the codeword $C_{posSwap}$ might become a continuer. Therefore, if the codeword following the $C_{sChange}$ tuple is a suffix of the codeword associated to a search pattern, a false match could be notified. Fortunately, the number of changes of $s$ is so small (under 4,465 in our experiments) that adding this extra byte does not noticeably worsen the compression ratio.

### 4.3 Evolution of swaps and changes on *s*

Determining the overhead of DLSCDC with respect to DSCDC involves studying two aspects: the number of changes of the parameter $s$, and the number of swaps.

We verified empirically that the number of changes on $s$ is very small. During the compression of the corpus ALL, described in Section 5, there are $4,465$ changes on $s$. In general, at the beginning of the compression process, the number of changes on $s$ grows fast, and then the slope of the curve becomes less steep. In the ZIFF corpus, from the 3,519 changes on $s$ in total, 20% of the changes occur when the first 10% of the text is processed. After this, the growth follows fairly well the model $y = a \cdot T$ (with a rather small constant $a$), where $y$ is the number of changes on $s$ produced after processing $T$ bytes of text. This can be seen in Figure 8 (a), which shows the number of changes on $s$ as the ZIFF corpus is compressed.

On the ZIFF corpus, most of the 4,570 swaps between codewords of 1 and 2 bytes appear during the processing of the first 10% of the file, as it is shown in Figure 8 (b), and as expected, most of the swaps (75,197) occur between codewords of 2 and 3 bytes (see Figure 8 (c)).

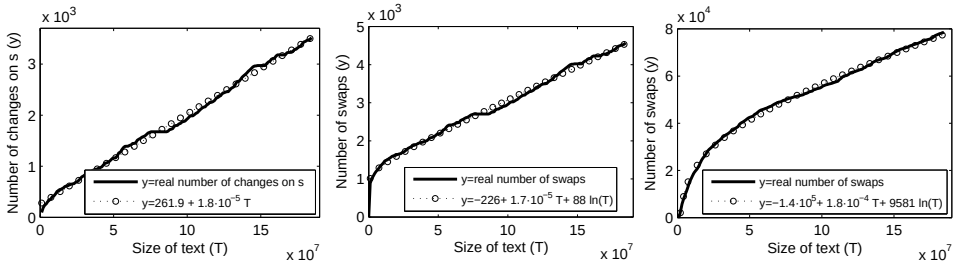



Fig. 8. (a) Changes on $s$, swaps of (b) 1 and 2 byte codewords, (c) 2 and 3 byte codewords.

From the 4,570 swaps between codewords of 1 and 2 bytes, 3,519 swaps are directly attributable to changes of $s$. To cope with these extra swaps, in our least square fittings, we add an additional linear factor to our logarithmic model, obtaining consistent results (see Figure 8 (b)).

Regarding the words around $W_2^s$, as shown, each change on $s$ could leave several codewords with incorrect length (e.g., 124 if changing from $s = 190$ to $s = 191$). Let us consider again the scenario depicted in Figure 5. After a decrease of $s$ from 132 to 131, words `"galaxy"` and `"Luke"` remain with codewords of 3 bytes, although they deserve 2-byte codewords. However, the next occurrence of any of these words can produce a regular swap. For example, let us consider a new occurrence of `"galaxy"`: after increasing its frequency, it is swapped with the word `"ago"`, since this is the first word in the ranked vocabulary with frequency 8. Figure 9 (a) shows the state of the vocabulary after this swap. Now the sender compares $C_{16507}$, the codeword of `"galaxy"`, with $C_{16499}$, the codeword of `"ago"`, and since $C_{16499}$ is encoded with 2 bytes and $C_{16507}$ with 3, the compressor issues a new swap, which exchanges the codewords of `"galaxy"` and `"ago"`. The new state of the vocabulary at the sender is shown in Figure 9 (b). The incorrect length codeword of 3 bytes, previously assigned to `"galaxy"`, is now assigned to `"ago"` (that still ought to have a 2-byte codeword), yet `"galaxy"` is now more probable and therefore deserves more the 2-byte codeword.

Again, in order to handle those extra swaps, we use the same model with linear and logarithmic terms, obtaining good fittings, as it can be seen in Figure 8 (c).

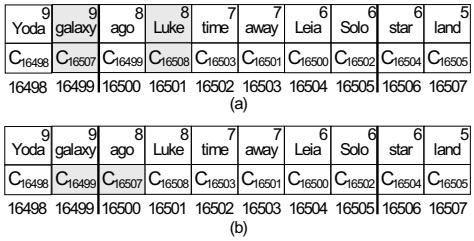



Fig. 9. Scenario after a new occurrence of the word `"galaxy"` with $s = 131$.

## 5. EXPERIMENTAL RESULTS

We used three text collections of different sizes. The small and medium size collections are the Calgary corpus[8] (CALG) and the Ziff Data 1989-1990 (ZIFF) collection from TREC-2[9], respectively. To obtain the largest collection (ALL), we aggregated the Calgary corpus, the ZIFF corpus, and several texts from TREC-2 and TREC-4, namely AP Newswire 1988, Congressional Record 1993 and Financial Times 1991 to 1994. We used the spaceless word model [Moura et al. 1998] to create the vocabulary, that is, if a word was followed by a single space, we just encoded the word, otherwise both the word and the separator were encoded.

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. We measure CPU user time in seconds.

[8] `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`.
[9] `http://trec.nist.gov`.

### 5.1 Compression ratio and compression/decompression speed

We split this empirical comparison into two sets. First, we compare DLETDC and DLSCDC against general-purpose compressors, including: *Gzip*[10] (a Ziv-Lempel compressor with performance very similar to *zip*) [Ziv and Lempel 1977], *Bzip2*[11] (a block sorting compressor) [Burrows and Wheeler 1994], and *ppmdi*[12] (a compressor using an *arithmetic encoder* coupled with a $k$-order modeler). Although these compressors use very different principles, we compare to them because they provide a measure of the effectiveness of DLETDC and DLSCDC from an end-user's point of view. In the case of *Gzip* three compression options were compared: best compression ('-9'), default compression ('-6'), and fastest compression ('-1').

To measure the effectiveness of the coding scheme exclusively, the second set of experiments compares DLETDC, DLSCDC, and DLSCDC$^{Tv}$ against other techniques that also use word-based modelling. As adaptive compressors, we included DETDC, DSCDC, and our own implementation of a word-based byte-oriented Dynamic Plain Huffman (DPH)[13]. Given that one of the features of DLETDC and DLSCDC is their search capability, to have qualified competitors in this category, we included the semistatic compressors ETDC and SCDC. Finally, as a baseline, we added *Vbyte*, a dynamic word-based compressor that uses the same coding scheme as ETDC, but does not use any statistical information of the text in order to obtain a good compression ratio. *Vbyte* simply assigns the first codeword to the first word in the text, the second codeword to the second distinct word in the text, and so on.

In any case, compression times consider that compressors are fed with the text in plain form. In the same way, decompression times include the complete process of recovering the original text. Finally, we recall that our compression ratios give the size of the compressed file as a percentage of the original size in plain form (text) and the compressed file size includes the size of its model.

*General-purpose set.* The results of this first experiment are given in Table I.

| | Corpus | Size(Mb) | Gzip-1 | Gzip | Gzip-9 | DLETDC | DLSCDC | ppmdi | Bzip2 |
|---|---|---|---|---|---|---|---|---|---|
| (a) | CALG | 2.0 | 43.53 | 36.95 | 36.84 | 50.16 | 51.82 | 26.39 | 28.92 |
| | ZIFF | 176.6 | 40.34 | 34.55 | 34.47 | 33.88 | 33.47 | 23.04 | 25.67 |
| | ALL | 1,030.7 | 41.31 | 35.09 | 35.00 | 33.69 | 33.19 | 24.21 | 25.98 |
| (b) | CALG | | 0.13 | 0.34 | 0.42 | 0.11 | 0.12 | 1.24 | 0.77 |
| | ZIFF | | 9.47 | 27.04 | 31.40 | 9.17 | 10.18 | 98.81 | 63.02 |
| | ALL | | 56.76 | 160.01 | 191.30 | 58.45 | 64.13 | 599.15 | 375.55 |
| (c) | CALG | | 0.06 | 0.04 | 0.05 | 0.03 | 0.04 | 1.31 | 0.30 |
| | ZIFF | | 3.84 | 3.38 | 3.36 | 2.46 | 2.44 | 102.85 | 25.71 |
| | ALL | | 23.12 | 21.05 | 20.98 | 16.27 | 15.86 | 631.09 | 156.18 |

Table I. DLETDC and DLSCDC against general-purpose compressors: (a) Compression Ratio (in percentage), (b) compression and (c) decompression times (in seconds).

It can be seen that *ppmdi* and *Bzip2* yield the best (23–26%) compression ratios. Our techniques achieve 33–34%, being slightly better than the most effective *Gzip*

[10] `http://www.gnu.org`.
[11] `http://www.bzip.org`.
[12] `http://pizzachili.dcc.uchile.cl`.
[13] `http://rosalia.dc.fi.udc.es/codes/`.

variant. The exception is the small collections (Calgary), where the vocabulary size is still significant compared to the text size.

With respect to compression times, DLETDC, DLSCDC, and *Gzip -1* obtain the best results, near 1 min/Gbyte, yet *Gzip -1* is very ineffective in compression ratios (40–41%). Both *Bzip2* and *ppmdi*, which achieve the best compression, are significantly (6–10 times) slower than the others. The *Gzip* variants that compete in compression ratios with our dynamic compressors are 2.5–3 times slower.

Finally, regarding decompression times, we remark that *Gzip* is a very efficient technique; in fact this is one of its main strengths. However, our lightweight techniques are more than 20% faster than any *Gzip* variant, and 10–40 times faster than *ppmdi* or *Bzip2*.

This shows that DLETDC and DLSCDC offer very attractive tradeoffs for dynamic natural language compression. On the other hand, one can argue that DLETDC and DLSCDC suffer from a higher memory consumption due to their use of a word-based model. While the amount of the memory used by *Gzip*, *ppmdi* and *bzip2* is bounded, DLETDC and DLSCDC must keep the vocabulary (i.e., the model) in memory.

We verified how serious is the problem in practice, in our setup. During the compression of the ALL corpus (1 Gbyte of text), DLETDC and DLSCDC consumed around 90 Mbytes, *Gzip* 720 Kbytes, *ppmdi* 10 Mbytes, and *bzip2* 7.8 Mbytes (using default options when applied). Taking into account that DLETDC and DLSCDC compressors are designed to run in a normal desktop computer (the sender), 90 Mbytes of memory consumption is perfectly reasonable.

Memory usage for decompression is of special interest, as DLETDC and DLSCDC decompressors should run over weaker machines such as handheld devices. During the decompression of the ALL corpus, DLETDC and DLSCDC consumed 16.5 Mbytes, in comparison with 520 Kbytes of *Gzip*, 10 Mbytes of *ppmdi*, and 5 Mbytes of *bzip2*. Yet, current PDA devices usually have around 128-256 Mbytes of memory, therefore it is perfectly realistic to assume that they will have enough memory to decompress any compressed text, independently of its size.

In general, by Heaps' law, the size of the vocabulary is of the form $v = K \cdot T^{\beta}$, $0 < \beta < 1$, which implies that the model grows sublinearly with the text size, as seen in Figure 4(c). In our experiments, as well as in many others [Baeza-Yates and Ribeiro-Neto 1999], $v$ is close to $O(\sqrt{n})$. Given that the space requirement of DLETDC and DLSCDC is $O(v)$ integers with modest constants[14], we expect that the memory requirements for compression and decompression will be small enough to allow DLETDC and DLSCDC handle large texts in current computer memories without any problem. Indeed, many inverted indexes assume that similar vocabulary structures fit in main memory [Baeza-Yates and Ribeiro-Neto 1999].

Another side of the coin is whether those general-purpose compressors can take advantage of more memory, that is, how would be the comparison if all used the same memory. With respect to the $k$-th order statistical compressors, we forced *ppmdi* to use more memory[15]. By using around 20 Mbytes (instead of 10 Mbytes), the compression ratio improves to 22.34%, yet at the expense of even worse com-

[14]The vocabulary size is $O(v)$. Fariña [2005] shows how to keep array *top* using $O(v)$ integers.
[15]In the case of *bzip2*, the use of option *-9* did not improve the default results.

pression times (636.57 seconds on corpus ALL). Thus, as expected, the gaps in compression ratio and in compression/decompression times just widen as *ppmdi* uses more memory.

The case of Ziv-Lempel compressors is more interesting, as we beat *Gzip* in compression ratio and time simultaneously. We tested software *p7zip*[16], which allows using a buffer of arbitrary length. By using a buffer of 32 Mbytes (instead of 32 Kbytes) and arithmetic encoding of the pointers, *p7zip* consumes 192 Mbytes when compressing corpus ALL, and 17.3 Mbytes during decompression. Its compression ratios (22.81% in corpus ALL) are clearly better than those of DLETDC and DLSCDC. Yet, compression times are now much worse, 1939.4 seconds for compressing and 54.03 seconds for decompressing corpus ALL. The result is now similar to $k$-th order compressors: it achieves better compression ratio at the expense of much worse compression and decompression times.

As a conclusion, if we provide the general-purpose compressors with more memory, to match ours, they will achieve better compression ratio than DLETC and DLSCDC, at the expense of significantly higher times. So if we want pure compression ratio, DLETDC and DLSCDC are not the best choice. However, if we want fast real-time compression and decompression with efficient direct search capabilities (the general-purpose compressors are not efficiently searchable), and reasonable compression ratios, then DLETDC and DLSCDC are a good alternative.

*Word-based compressors.* Table II shows the results of this second comparison.

| | Corpus | DETDC | DSCDC | DPH | Vbyte | DLETDC | DLSCDC | DLSCDC$^{Tv}$ | ETDC | SCDC |
|---|---|---|---|---|---|---|---|---|---|---|
| (a) | CALG | 47.73 | 46.81 | 46.55 | 55.18 | 50.16 | 51.82 | 49.60 | 47.40 | 46.61 |
| | ZIFF | 33.79 | 33.08 | 32.90 | 38.03 | 33.88 | 33.47 | 33.17 | 33.77 | 33.06 |
| | ALL | 33.66 | 33.03 | 32.85 | 45.00 | 33.69 | 33.19 | – | 33.66 | 33.02 |
| (b) | CALG | 0.09 | 0.11 | 0.12 | 0.07 | 0.11 | 0.12 | 0.11 | 0.16 | 0.16 |
| | ZIFF | 8.53 | 9.64 | 10.92 | 6.32 | 9.17 | 10.18 | 9.34 | 11.89 | 11.75 |
| | ALL | 55.31 | 61.35 | 71.54 | 39.55 | 58.45 | 64.13 | – | 75.58 | 75.20 |
| (c) | CALG | 0.04 | 0.05 | 0.07 | 0.03 | 0.03 | 0.04 | 0.03 | 0.03 | 0.04 |
| | ZIFF | 3.82 | 4.44 | 7.11 | 2.48 | 2.46 | 2.44 | 2.38 | 2.25 | 2.55 |
| | ALL | 25.27 | 28.62 | 50.82 | 17.45 | 16.27 | 15.86 | – | 14.56 | 15.08 |

Table II. DLETDC and DLSCDC against word-based compressors: (a) Compression Ratio (in percentage), (b) compression and (c) decompression times (in seconds).

In compression ratio, DPH, which generates optimal prefix-free byte-codes, obtains the best results. The opposite case is *Vbyte*, which obtains the worst results due to poor modelling. The artifacts required to produce a lightweight dynamic dense code compressor do not introduce differences of more than 0.4 percentage points with respect to any other dense code variant, if we except the small collection (where differences can reach 5 percentage points).

*Vbyte* simplicity makes it unbeatable in compression speed. Due to the management of swaps, DLETDC and DLSCDC$^{Tv}$ are slightly slower than DETDC. Being around 10% slower than DLETDC, DLSCDC pays also for the efforts needed to maintain the parameter $s$ well tuned. DPH, ETDC and SCDC also obtain good

[16] `http://p7zip.sourceforge.net/`

performance, but they are overcome by the dynamic dense compressors, in the case of DPH due to the complexity of maintaining a well-formed Huffman tree dynamically, and in the case of ETDC and SCDC due to the two passes over the original text.

With respect to decompression time, DPH is about three times slower than DLETDC and DLSCDC, which are on a par with the very fast semistatic dense codes and the simple *Vbyte*.

To sum up, DLETDC is easier to program, compresses more, and compresses and decompresses faster than *Gzip*. This is enough by itself to make DLETDC an interesting choice for dynamic compression of natural language texts. Yet, DLETDC has other relevant features, as we show in the next section. DLSCDC$^{Tv}$ goes even further, improving the compression ratio obtained by DLETDC while obtaining results similar to those of DLETDC at compression and decompression speed. The same is applicable to DLSCDC, which is a bit slower at compression, but is able to overcome DLETDC at decompression speed and compression ratio. We note that DLSCDC is almost twice as fast as DSCDC for decompression, hence well deserving the tag "lightweight".

### 5.2 Searching compressed and uncompressed text

We performed single-pattern and multi-pattern searches for patterns chosen at random over corpus ALL. We carry out two kinds of comparisons. The first is aimed at showing how much is gained or lost, at search time, due to having the text in compressed form. Here we compare searching text compressed with DLETDC versus searching uncompressed text, either in plain or tokenized form (details later). The second kind of comparison is among different encoders that achieve comparable compression ratios, in order to show how our new compressors perform, at text searching, with respect to similar compressors. Here we also include the best searcher on text compressed with a general-purpose compressor.

We first describe the search techniques that work over text compressed using ETDC, DLETDC, DLSCDC, DETDC, and *Gzip -9*. To search text compressed with ETDC, we use our own implementations of *Horspool* and *Set-Horspool* algorithms [Horspool 1980; Navarro and Raffinot 2002]: *Hoorpool* for single-pattern searches, and *Set-Horspool* for multi-pattern searches. *Horspool* is the best option to search for a pattern on a relatively large alphabet ($|\Sigma| = 256$ in the case of ETDC) [Navarro and Raffinot 2002]. In the case of multi-pattern searches, a *Wu-Manber*-based searcher [Wu and Manber 1994] might seem the best choice [Navarro and Raffinot 2002]. Basically, *Wu-Manber* is a *Set-Horspool* algorithm that inspects blocks of characters instead of individual ones, and computes shift values for such blocks. When searching uncompressed text, it typically uses blocks of size $B = 2$–$3$ characters. The recommended value is $B = \log_{|\Sigma|}(2 \cdot K \cdot \ell_{min})$, where $|\Sigma|$ is the alphabet size, $K$ is the number of patterns sought, and $\ell_{min}$ is the length of the shortest pattern. In Dense Codes, it turns out that *Wu-Manber* would use $|B| = 1$, boiling down to the simpler *Set-Horspool* algorithm.

The simplest way of searching text compressed with DLETDC, DLSCDC, and DETDC, is to mark the vocabulary words that match the pattern and then simulate the decompression processes, reporting occurrences of marked words instead of emitting all the source words. Since all of the bytes in the compressed file are

processed, this variant is called *all-bytes* (or just *all*).

As shown in Sections 3.2 and 4.2, *Set-Horspool* algorithm can be used to perform Boyer-Moore-type searches over text compressed with DLETDC and DLSCDC. These searchers are tagged (*S-H*) in the experiments. Recall that the simple *Horspool* cannot be used here, as we must always search for $C_{swap}$, in addition to the true patterns.

In DLETDC, DLSCDC, and 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. Those variants are tagged *+dec* in the experiments.

As explained, we compare with the best way to search text compressed with a general-purpose compressor. As far as we know, the best alternative in practice is *LZgrep* [Navarro and Tarhio 2005]. We included in our comparison the authors' implementation[17]. *LZgrep* permits direct searching text compressed with LZ77 (i.e., *Gzip*) and LZW (i.e., *Compress*) formats [Ziv and Lempel 1977; 1978] faster than performing decompression plus searching. *LZgrep* is fastest on *Gzip -9*, thus we used this format. We note that *LZgrep* did not search for more than 100 patterns.

On the other hand, three different algorithms were tested to search the uncompressed text: our own implementations of *Horspool* and *Set-Horspool* algorithms, and the *agrep*[18] software [Wu and Manber 1992a; 1992b], a fast pattern-matching tool which allows, among other things, searching a text for multiple patterns. According to Navarro and Raffinot [2002], the best option for the single-pattern search on English text is *Horspool*. For multi-pattern searches, the best option is the *Wu-Manber* algorithm, which is included in our experiment by using *agrep*.

*Agrep* searches return chunks of text containing one or more searched patterns. The default chunk is a line. When traversing a chunk, if *agrep* finds a search pattern, it skips the processing of the rest of the chunk. This appreciably distorts the comparison against the rest of the searchers. To avoid this pernicious effect, we performed the searches over a modified version of the text obtained by removing all the pattern occurrences from it, and then scaled the results. More precisely, we computed the text $T'$ that is obtained by removing all pattern occurrences from the original text (ALL corpus). Then we ran *agrep -s* over $T'$ and we scaled the resulting times assuming that $|T'| = |ALL|$. This shows essentially the same statistics and reflects more accurately the real search cost of *agrep*.

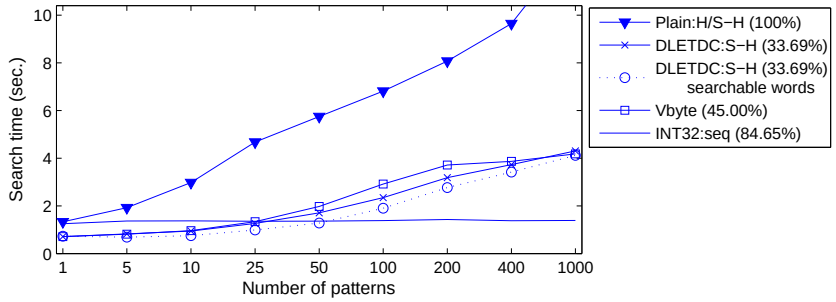
Finally, following the *integer-on-integer searching* approach presented by Culpepper and Moffat [2006], we show the effect of searching a tokenized text (that is, words are mapped to integers and the integer sequence is scanned instead of the source text). In our experiments this type of search is denoted *INT32:seq*. This technique is very easy to implement and extremely fast, as the symbols (words) have now fixed length. We note this is not a compression method, but rather serves to illustrate how fast the search can be if we do not use variable-length encoding of the source words in order to achieve compression. As an intermediate alternative,

[17] `http://www.dcc.uchile.cl/gnavarro/software/lzgrep.tar.gz`.
[18] `ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z`

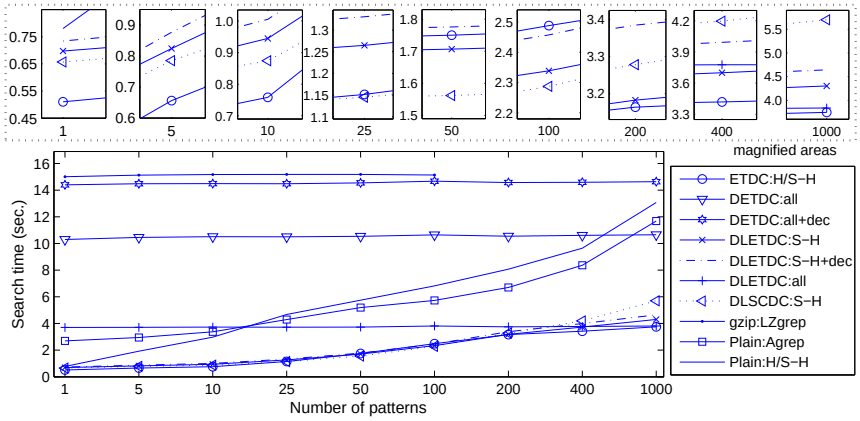
we also compare the search over *Vbyte* using *Horspool* for single-pattern searches and *Set-Horspool* for multi-pattern searches.

To choose the search patterns, we considered the vocabulary of corpus ALL, from which we removed both the *stopwords* (prepositions, articles, etc.) and the separators (sequences of non-alphanumerical characters). Then, following the model where each word is sought with uniform probability [Moura et al. 2000], we extracted 100 sets with $K$ words of length $L$. We considered lengths $L = 5, 10$, and $> 10$, and for each length we measured $K = 1$, 5, 10, 35, 50, 100, 200, 400, and 1000 patterns. We only show the case of lengths $L > 10$, which gives an advantage to the uncompressed text searchers, whereas the others are mostly insensitive to $L$.

Figure 10(a) shows the first experiment, where we compared DLETDC*:S-H* with searches over plain text, *Vbyte*, and *INT32:seq*. This gives a clear picture of the search speed of DLETDC with respect to the alternative of not compressing the text with competitive techniques.




(a) DLETDC against not compressing text with competitive techniques.




(b) Dense code variants against *Agrep* and *S-H* over plain text, and *LZgrep*.

Fig. 10. Searching for multiple patterns.

As expected from previous work [Ziviani et al. 2000], compressed text searching is several times faster than searching the plain uncompressed text, which was

described in that work as a "win-win situation": compressing the text with some word-based modelling methods achieves a large gain in space and search speed simultaneously. Now, the comparison with *INT32:seq* shows that this gain is mostly due to the word tokenization that is implicit in the modelling. Searching the tokenized text, as a sequence of integers, is extremely fast, even if the "compression ratio" it achieves is just around 85%. Still, when searching for up to 25 patterns, the further compression achieved by assigning variable-length byte codes to the tokens further improves search times. After that point, the *Set-Horspool* shifts become short enough on average to degrade the performance well beyond that achieved by *INT32:seq*.

The performance of *Vbyte* is rather similar to DLETDC*:S-H*, and slightly better, again, when searching for many patterns. In fact, this is unavoidable: A better encoder will find the way to assign shorter codewords to more text words than a worse encoder. Thus if those words are sought, the *Horspool* shifts will be shorter and the search will be slower. To illustrate this point, we also show in Figure 10(a) the performance of DLETDC*:S-H* when the 128 most frequent words (i.e., those receiving a 1-byte codeword) are forbidden from the search (called "searchable words" in the figure). Now the performance is always better than *Vbyte*, and better than *INT32:seq* for up to 50 patterns.

Figure 10(b) displays the results of the second experiment. We include *LZgrep*, all the dense code variants and the plain text searchers. DLETDC*:all+dec* and DLSCDC*:S-H+dec* are not displayed to avoid overloading the figure, given that their times are practically the same as those of their corresponding version without the *+dec* option. In the same way, we removed the data from DLSCDC*:all+dec* and DLSCDC*:all*, as their times are very close to those of DLETDC*:all*.

DETDC and DSCDC introduced several improvements on previous compressors in the dynamic scenario [Brisaboa et al. 2008]. Regarding searches, DETDC obtained slightly better times than *LZgrep*, its main competitor. At that point of the state-of-the-art, DETDC and *LZgrep* already achieved search times that made them preferable to *decompression and searching*. Yet, as shown in the figure, they were far from being faster than uncompressed text searching.

However, by breaking the usual symmetry of the dynamic compressors, the lightweight dynamic dense compressors are now able to overcome the traditional adaptive compressors by far. Even the *:all* versions of our searchers are 2.5 times faster than the best DETDC searcher (note that DETDC is intrinsically of type *:all*). Moreover, using *Set-Horspool*, lightweight adaptive dense compressors obtain search times that are, at worst, 25% slower than those obtained by the very fast semistatic compressors, and can be even faster when the number of patterns is not very high. Finally, we emphasize that searching text compressed with DLETDC and DLSCDC is, for the first time in the dynamic scenario, faster than searching the uncompressed text. In the case of single-pattern searches, DLETDC and DLSCDC are from 17%, in the case of long patterns where *Horspool* over plain text can perform longer shifts, up to 4 times faster than searching plain text. When performing multi-pattern searches, DLETDC is usually twice as fast as (and up to 7 times, on short patterns) *Agrep* and *Set-Horspool*.

Obviously, being faster than plain text searchers, lightweight dense techniques

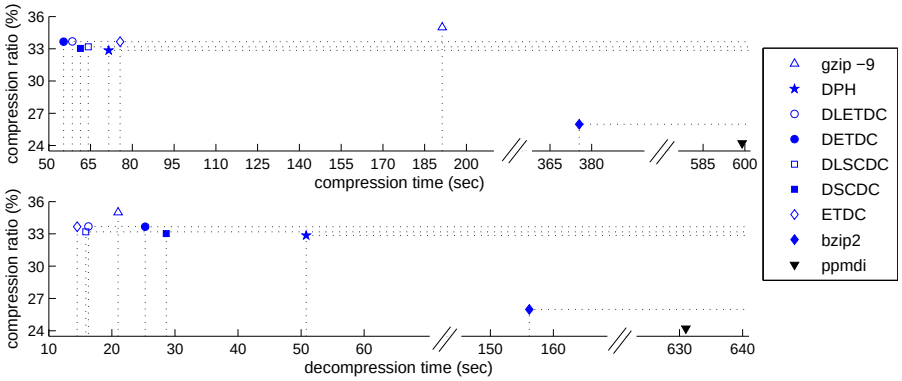



Fig. 11. Space/time trade-offs among dynamic techniques, on corpus ALL, related to compression/decompression time.

also beat previous searchers over dynamic compressed text. DLETDC:*S-H* is 4 to 18 times faster than *LZgrep*.

## 6. CONCLUSIONS

We have developed new adaptive compressors for natural language text (DLETDC and DLSCDC) that use dense coding coupled with a word-based model. Their main conceptual innovation is to break the compressor/decompressor symmetry that is usual in statistical adaptive compression. Now the sender keeps track of the statistical model and informs the receiver of the changes in codeword assignments. This is coupled with a mechanism to carry out only the changes that are absolutely necessary to maintain the compression effectiveness. The decompressor is much lighter and can be run over weaker processors such as handheld devices.

The new compressors are well-suited for addressing the problem of efficient transmission of natural language text documents. They retain several desirable features of previous dynamic text compressors based on dense codes: simplicity, good compression ratios (around 32-36%), and fast compression. The new techniques become much faster at decompression than their predecessors, and stand out as attractive space/time trade-offs. In particular, they overcome in all aspects the popular Ziv-Lempel compressors, for sufficiently large natural language texts.

In addition, the compressed text can be searched with much less work than that of decompressing it. The search is almost as fast as on semistatic compression formats and much faster than the uncompressed text, breaking a long-standing assumption that adaptive statistical methods would never be efficiently searchable.

Figures 11 and 12 show trade-offs between compression ratio and efficiency on different tasks, to illustrate the results achieved by our new techniques.

Future work involves trying to use these dynamic techniques to cope with compression of text collections that grow over time. Semistatic methods require to rebuild the compressed text from time to time, so as to include new words that appeared and update the statistics. With a dynamic code this updating is automatic, yet it is not possible to access a document at random from the collection. Some preliminary works [Brisaboa et al. 2005a] shows that a combination of techniques

could give good results.

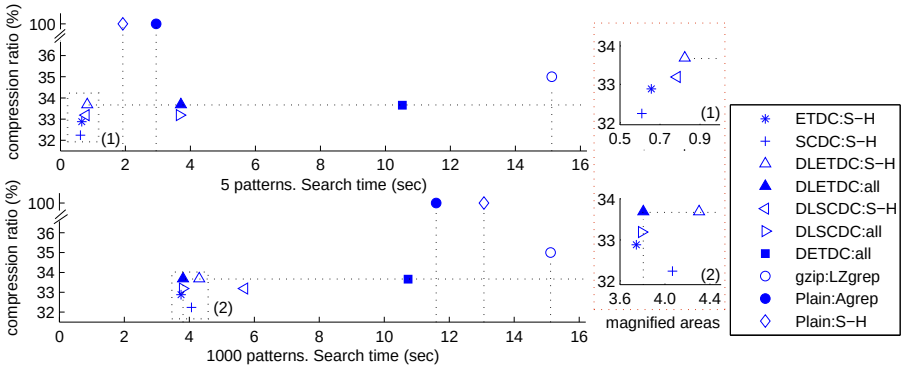



Fig. 12. Space/search time trade-offs among dynamic techniques, on corpus ALL, related to search time for patterns of length $> 10$.

## REFERENCES


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

Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. 1986. A locally adaptive data compression scheme. *Communications of the ACM 29,* 4, 320–330.

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

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

Brisaboa, N., Fariña, A., Navarro, G., and Paramá, J. 2005b. 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)*. ACM Press, 234–241.

Brisaboa, N., Fariña, A., Navarro, G., and Paramá, J. 2007. Lightweight natural language text compression. *Information Retrieval 10,* 1, 1–33.

Brisaboa, N., Fariña, A., Navarro, G., and Paramá, J. 2008. New adaptive compressors for natural language text. *Software Practice and Experience 38*, 1429–1450.

Burrows, M. and Wheeler, D. J. 1994. A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation.

Cleary, J. G. and Witten, I. H. 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications 32,* 4 (Apr.), 396–402.

Culpepper, J. and Moffat, A. 2005. Enhanced byte codes with restricted prefix properties. In *Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE'05)*. LNCS 3772. Springer-Verlag, 1–12.

Culpepper, J. S. and Moffat, A. 2006. Phrase-based pattern matching in compressed text. In *SPIRE*, F. Crestani, P. Ferragina, and M. Sanderson, Eds. Lecture Notes in Computer Science, vol. 4209. Springer, 337–345.

Fariña, A. 2005. New compression codes for text databases. Ph.D. thesis, Database Laboratory, University of A Coruña, Spain. Available at `http://coba.dc.fi.udc.es/~fari/phd/`.

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

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

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

Moura, E., Navarro, G., Ziviani, N., and Baeza-Yates, R. 1998. Fast searching on compressed text allowing errors. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'98).* ACM Press, 298–306.

Moura, E., Navarro, G., Ziviani, N., and Baeza-Yates, R. 2000. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems 18,* 2, 113–139.

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

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

Navarro, G. and Tarhio, J. 2005. LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text. *Software Practice and Experience 35,* 12, 1107–1130. Relevant software available at `http://www.dcc.uchile.cl/~gnavarro/software/lzgrep.tar.gz`.

Turpin, A. and Moffat, A. 1997a. Efficient approximate adaptive coding. In *DCC '97: Proceedings of the Conference on Data Compression.* IEEE Computer Society, 357–366.

Turpin, A. and Moffat, A. 1997b. Fast file search using text compression. In *Proceedings of the 20th Australian Computer Science Conference (ACSC'97).* 1–8.

Williams, H. E. and Zobel, J. 1999. Compressing integers for fast file access. *The Computer Journal 42,* 3, 193–201.

Wu, S. and Manber, U. 1992a. Agrep – a fast approximate pattern-matching tool. In *Proceedings of the USENIX Winter 1992 Technical Conference.* 153–162.

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

Wu, S. and Manber, U. 1994. A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ.

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

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

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

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