Compressing dynamic text collections via phrase-based coding *

Nieves R. Brisaboa[1], Antonio Fariña[1], Gonzalo Navarro[2] and José R. Paramá[1]

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

Abstract. We present a new statistical compression method, which we call *Phrase Based Dense Code (PBDC)*, aimed at compressing large digital libraries. PBDC compresses the text collection to 30–32% of its original size, permits maintaining the text compressed all the time, and offers efficient on-line information retrieval services. The novelty of PBDC is that it supports continuous growing of the compressed text collection, by automatically adapting the vocabulary both to new words and to changes in the word frequency distribution, without degrading the compression ratio. Text compressed with PBDC can be searched directly without decompression, using fast Boyer-Moore algorithms. It is also possible to decompress arbitrary portions of the collection. Alternative compression methods oriented to information retrieval focus on static collections and thus are less well suited to digital libraries.
Keywords: Text Compression, Text Databases, Digital Libraries.

1 Introduction

Digital libraries can be thought of as a text collection plus a set of online text search and retrieval functionalities. In many cases of interest the collection grows over time, while in others it is static. On the other hand, text compression [2] aims at representing text using less space. Storing the text collection of a digital library in compressed form saves not only space, but more importantly, disk and network transmission time. In the last decades, CPU performance has been doubling every 18 months while disk access times have stayed basically unchanged. Thus it is worthwhile to pay compression and decompression overheads in exchange for reduced disk times.

There are several challenges, however, to offer digital library services over a compressed text collection. It should be possible to carry out efficiently the following tasks: (1) uncompress any portion of the collection; (2) accommodate new text that is added to the collection; (3) scan text portions searching for

* This work is partially supported by CYTED VII.19 RIBIDI Project. It is also funded in part (for the Spanish group) by MCyT (PGE and FEDER) grant(TIC2003-06593) and (for G. Navarro) by Fondecyt Grant 1-050493, Chile.

keywords. Task (1) is related to the need to display text documents in plain form to the user. Task (2) has to do with handling growing collections. Task (3) appears when searching the text collection for words or phrases of interest. This is usually faced by means of inverted indexes that permit locating the documents of interest without traversing the text. However, indexes take additional space that must be maintained at a minimum [16, 1, 12]. Hence it is customary that indexes do not have all the information necessary to solve all queries (in particular phrase and proximity queries) without accessing the text. Therefore, scanning text portions is usually necessary.

Two main approaches exist to text compression. *Adaptive* compression methods, such as the well-known Ziv-Lempel family [18, 19], learn the data distribution as they compress the text, continuously adapting their model of the distribution. The same model has to be learned and updated during decompression. *Semistatic* methods, such as Huffman [8], make a first pass over the text collection to build a model of the data, and compress the whole collection with the same model in a second pass.

The need of task (1) prevents the use of adaptive compression methods, as these require decompression to start at the beginning of the collection, which would make impossible to carry out local decompression efficiently. It would be possible to compress documents individually, but then compression ratios are poor because there is not enough time for the model to converge to the data distribution. This is unfortunate because adaptive compression methods would deal best with task (2), by simply appending the new text to the end of the collection and going on with the compression.

Task (3) can be faced by just decompressing the text and then searching it. A much more attractive alternative is to search the compressed text directly, without decompressing it. This saves CPU time because only a small fraction of the searched text will be finally shown to the user. Albeit there exist several techniques to search text compressed with adaptive methods (e.g. [14]), much more efficient methods have been devised for semistatic techniques. The essential reason is that one can compress the pattern and search the text for it, as its compressed form does not vary across the text. Adaptive methods, on the other hand, require keeping track of the model updates. In particular, it has been shown [5] that variants of Huffman permit searching the text up to 8 times faster than over the plain text (not even counting the time to decompress).

Classical Huffman is an unpopular choice for text compression because of its poor compression ratios. However, if the source symbols are taken to be the whole words rather than the characters [9], compression ratios (the size of the compressed text as a fraction of the original text) improve to 25%-30%, which is rather competitive. The reason is that the distribution of words is much more biased than that of characters, thanks to Zipf's Law [17]. Moreover, the source alphabet and the collection vocabulary are the same, which simplifies integration with inverted indexes [16, 12]. The Huffman variants that yield those good results in search times [5] are actually word-based. Moreover, the output is a sequence of bytes rather than bits. This worsens compression ratio a bit (it gets close to

30%), but in exchange decompression is much faster. Some variants of the format, such as *Tagged Huffman*, get compression ratios closer to 35% but ensure that the text is efficiently searchable with any text search algorithm.

The above comprises a good solution for a *static* digital library to maintain the text in compressed form, by using a semistatic compression method like word-based Huffman or a variant thereof. The compressed text takes 25%-35% of the original size, the index adds 5%-15% to this, and the search is *faster* than without compression. Hence space *and* time are saved simultaneously.

The situation is more complicated when growing collections are to be handled. Semistatic methods do not work because they rely on a global model, so in principle they need to recompress the whole text collection again. The only proposal to handle this problem [10, 11] has been to build a semistatic model on the current text collection and then use it as is for the new text that arrives (new words are handled somehow), with no or very sporadic global re-compressions. This works reasonably well when the growing collection stays homogeneous, but this is not the case of most digital libraries (see next section).

In this paper, we present a modification of a statistical semistatic method, ETDC [4] (see next section), adapting it to deal with growing collections. The idea is to combine statistical compression (where variable-length codewords are assigned to fixed-length source symbols) with dictionary-based compression (where varying-length source symbols are concatenated and assigned fixed-length codewords). The resulting new method, called *Phrase Based Dense Code (PBDC)*, satisfies all the requirements of growing digital libraries and maintains the same efficiency obtained by semistatic methods. In particular, PBDC: 1)obtains good compression ratios over natural language text; 2)uses a unique vocabulary for the whole collection; 3)permits continuous increments of the compressed collection by automatically adapting the vocabulary both to new words and to changes in the word frequency distribution without degrading the compression ratio; 4)supports direct search without decompressing the text using any string matching algorithm; 5)is easily and efficiently decompressible at any arbitrary section of the text without need of performing the decompression of the whole document; and 6)uses structures easy to assemble to those of the classical inverted indexes that any digital library needs.

We present empirical data measuring the efficiency of PBDC in compression ratio, compression and decompression speed and direct search capabilities.

2 Related Work

2.1 Compressing Growing Collections

Some authors have proposed the use of semistatic statistical compression techniques such as Plain Huffman or Tagged Huffman over a first part of the collection and use the obtained vocabulary to compress the remaining text of the collection [10, 11]. That is, they propose to use the same old codewords for that words in the new text that already exist in the vocabulary, and compute new

codewords for the new ones. To manage the new words that can appear, different alternatives were proposed. For example, in [10] new words are inserted at the end of the vocabulary, and new codewords are generated for them. However, changes in the word frequencies are not taken into account. In [11], new words are not inserted into the vocabulary. When one appears, it is introduced in the compressed text and marked with a special previously defined codeword. To save space those new words are compressed with a static character-based Huffman code. Again, changes in the word frequencies are not taken into account.

In both cases [10, 11], authors argue that the loss of compression ratio is not significant. For example in [11], some experiments were performed over the AP archive of the TREC collection. This archive occupies 200MB. Compressing the whole file with Tagged Huffman a 31.16% compression ratio is achieved. When only a 10% of the file was compressed with Tagged Huffman and then the obtained vocabulary was used to compress the rest of the file, compressing new words with a static character oriented Huffman, the compression ratio raised to 32%. When the initial compression was performed over the 40% of the file the compression ratio became 31.5%.

These experiments were done over experimental corpora that, in our opinion, do not reproduce the situation that can be found in real digital libraries. In digital libraries, the amount of digitized text grows year after year and therefore the initial portion of the corpus, used to compute the initial vocabulary, becomes smaller and smaller. On the other hand, the vocabulary is expected not to be so homogeneous as it is inside one specific collection such as AP. In real life, new words appear and become very frequent. For example, if we think in a digital library of journals, names of politicians, artists, etc. appear when they get a noticeable position and later, after some years of having high frequency, they may disappear. The same applies to other words, names of places, events, or new technologies. For example, words such as *Web* or *Internet* have not a significant frequency in journals some years ago. This constant appearance and/or changes in frequency of words in real life could produce larger loss in compression ratio than those found in [11, 10].

2.2 End Tagged Dense Code

End-Tagged Dense Code (ETDC) [4] is a semistatic compression technique, and it is the basis of the *Phrase Based Dense Code (PBDC)* we present in this paper. ETDC is an improvement upon Tagged Huffman Code [5].

Let us call Plain Huffman Code the word-based Huffman code that assigns a sequence of bytes (rather than bits) to each word. In Tagged Huffman, the first bit of each byte is reserved to flag whether the byte is the first of its codeword. Hence, only 7 bits of each byte are used for the Huffman code. Note that the use of a Huffman code over the remaining 7 bits is mandatory, as the flag bit is not useful by itself to make the code a prefix code[1]. The tag bit permits

[1] In a prefix code, no codeword is a prefix of another, a property that ensures that the compressed text can be decoded as it is processed.

direct searching on the compressed text by just compressing the pattern and then running any classical string matching algorithm like Boyer-Moore [13]. On Plain Huffman this does not work, as the pattern could occur in the text not aligned to any codeword [5].

Instead of using a flag bit to signal the *beginning* of a codeword, ETDC signals the *end* of the codeword. That is, the highest bit of any codeword byte is 0 except for the last byte, where it is 1.

This change has surprising consequences. Now the flag bit is enough to ensure that the code is a prefix code regardless of the contents of the other 7 bits of each byte. To see this, consider two codewords X and Y, being X shorter than Y ($|X| < |Y|$). X cannot be a prefix of Y because the last byte of X has its flag bit in 1, while the $|X|$-th byte of Y has its flag bit in 0. Thanks to this change, there is no need at all to use Huffman coding in order to maintain a prefix code. Therefore, all possible combinations of bits can be used over the remaining 7 bits of each byte, producing a *dense* encoding. This yields a better compression ratio than Tagged Huffman while keeping all its good searching and decompression capabilities. On the other hand, ETDC is easier to build and faster in both compression and decompression.

In general, ETDC can be defined over symbols of b bits, although in this paper we focus on the byte-oriented version where $b = 8$.

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

That is, the first word is encoded as $\underline{1}0000000$, the second as $\underline{1}0000001$, until the 128^{th} as $\underline{1}1111111$. The 129^{th} word is coded as $\underline{0}0000000{:}\underline{1}0000000$, 130^{th} as $\underline{0}0000000{:}\underline{1}0000001$ and so on until the $(128^2 + 128)^{th}$ word $\underline{0}1111111{:}\underline{1}1111111$. Note that the code depends on the rank of the words, not on their actual frequency. As a result, only the sorted vocabulary must be stored with the compressed text for the decompressor to rebuild the model.

It is clear that the number of words encoded with 1, 2, 3 etc, bytes is fixed (specifically 128, 128^2, 128^3 and so on) and does not depend on the word frequency distribution. Generalizing, being k the number of bytes in each codeword ($k \ge 1$) words at positions i:

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

will be encoded with k bytes. These clear limits mark the change points in the codeword lengths and will be relevant in the PBDC that we present in this paper.

But not only the sequential procedure is available to assign codewords to the words. There are simple *encode* and *decode* procedures that can be efficiently implemented, because the codeword corresponding to symbol in position i is

obtained as the number x written in base 2^{b-1}, where $x = i - \frac{2^{(b-1)k} - 2^{b-1}}{2^{b-1}-1}$ and $k = \left\lfloor \frac{\log_2(2^{b-1} + (2^{b-1}-1)i)}{b-1} \right\rfloor$, and adding 2^{b-1} to the last digit.

Function *encode* obtains the codeword $C_i = encode(i)$ for a word at the i-th position in the ranked vocabulary. Function *decode* gets the position $i = decode(C_i)$ in the rank for a codeword C_i. Both functions take just $O(l)$ time, where $l = O(\log(i)/b)$ is the length in digits of codeword C_i. Those functions are efficiently implemented through just bit shifts and masking.

End-Tagged Dense Code is simpler, faster, and compresses 7% better than Tagged Huffman codes. In fact ETDC only produces an overhead of about 2% over Plain Huffman. On the other hand, since the last bytes of codewords are distinguished, ETDC has all the search capabilities of Tagged Huffman code. Empirical comparisons between ETDC and Huffman can be found in [4].

3 The Phrase Based Dense Code

PBDC is a hybrid approach that requires two phases. In the first phase, the initial corpus is compressed using ETDC, which produces the initial vocabulary. Then the PBDC algorithm is used to dynamically add each new document. An important property of PBDC is that the codeword↔word mapping will never change once it has been defined. That is, no matter what happens later, each word in the vocabulary will be always associated with the same and original codeword it was assigned.

From now on, we will call *phrases* to our input symbols. These phrases can be either just one word (as those in the initial vocabulary) or the concatenation of two or more words.

During the addition of new documents (second phase), we look for the longest known phrase that starts at the current position. For instance, if we read the word X, and X is already in the vocabulary, we will read the next word Y to create the phrase XY and we will check if phrase XY is also in the vocabulary. If it is, the next word Z will be read and concatenated to form phrase XYZ and so on. On the other hand, if XY is not in the vocabulary then X will be the longest known phrase starting at the current position.

When the longest known phrase at the current position is found, we compress it according to three different cases. Let us call αW the sequence of words starting at the current text position, so that α is the longest known phrase we found and W is the word that follows it (note that α might be the empty string ε).

New Phrase Case ***($\alpha = \varepsilon$).*** If the next input word W is not in the known vocabulary, then such one-word phrase W will be inserted in the vocabulary, its frequency will be set to one, and the next free codeword will be assigned to the new phrase W from now on. In addition, this new codeword will be used to compress this first occurrence of phrase W and compression will continue with the text that follows W.

No Change Case. Phrase $\alpha \neq \varepsilon$ is already in the vocabulary. The algorithm then increases its frequency by 1. If that new frequency corresponds to a

codeword of the same length as that already assigned to α, then phrase α will be compressed with its usual codeword. Compression will continue from word W, which has not yet been dealt with.

Concatenation Case. The interesting case arises when phrase $\alpha \neq \varepsilon$ is already in the vocabulary, and after increasing its frequency, it turns out that the new frequency corresponds to a codeword shorter than the one already assigned to α. In this case, α and W are concatenated to form a new longer phrase αW. This new phrase will be dealt with exactly as in the *New Phrase Case* (creating a new codeword for it, and so on), and compression will continue with the text that follows W.

Note that the idea is that, since we cannot assign a shorter codeword to a word or phrase that has increased its frequency, we opt for concatenating it with the word that follows it so that, instead of using shorter codewords, we compress more words with the codewords. This resembles the way Ziv-Lempel compression takes advantage of frequently occurring phrases in the input. Actually the algorithm has some similarity with LZ78 parsing [19].

3.1 Data Structures and Compression Procedure

The data structures used to compress, uncompress, and search, along with their functionality, are sketched in Figure 1, where the three cases are illustrated. The *vocabulary array* keeps each distinct word in the source text in a compact way (marking the end of each word with a terminator character).

The *Hash Table* is used during the compression and search process. This table keeps the source phrases αW by using two pointers in vector *phrases*. The first points to the slot in the hash table that keeps phrase α, while the second points to the position in the vocabulary vector where word W is stored. For instance, in Figure 1, phrase B is represented in slot 5, therefore the first pointer in vector *phrases* in the slot representing phrase BE (slot 7) points to slot 5, and the second pointer points to word E in the *vocabulary* array.

The hash table keeps in *freq* the phrase frequency and in *codeword* its codeword. It also maintains an array *codewordlist*, which is used only for searching and is explained in Section 3.3.

In the *Codewords Array*, each slot $i = decode(C_i)$ corresponds to codeword C_i. This array is used to decompress a document. Each slot stores a pointer to the slot in the *Hash Table* corresponding to the phrase encoded by C_i.

Let us assume that we process the first n words in the collection with ETDC, and then m new words that arrive using PBDC. The complexity of the first phase (ETDC) corresponds to: computing the vocabulary frequencies ($O(n)$), sorting the vocabulary of v distinct words ($O(v \log v)$), assigning a codeword to each word in the vocabulary ($O(v)$) and finally, compressing the text ($O(n)$). Since, empirically, $v = O(n^\beta)$ for $0 < \beta < 1$ [6], all the complexities add up $O(n)$.

The second phase (PBDC) costs $O(m)$, as we read each word at most twice (once to detect that it is not part of the next phrase, and once as the first word of the following phrase). Each new phrase requires $O(1)$ time to be dealt with. Thus, the complexity of the whole process is $O(n+m)$, linear in the whole text.

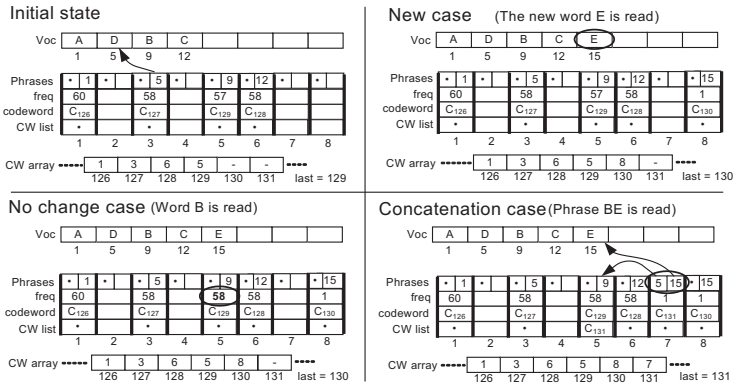

Fig. 1. Basic structures used and typical cases.

3.2 Deciding when to create a new phrase

There are different ways to know when a phrase α deserves a shorter codeword. For example, it is possible to keep the phrases sorted by frequency as in [3]. However, here we followed a statistical approach based on Chebyschev's inequality.

Let us call *group* i to the set of phrases with i-byte codewords. For each such group, we maintain the average $\hat{\mu_i}$ and standard deviation $\hat{\sigma_i}$ of the frequencies in the group. If we take the frequencies of group i as samples of a random variable X_i, then $\hat{\mu_i}$ and $\hat{\sigma_i}$ are the unbiased estimators of the mean μ_i and standard deviation σ_i of X_i. Chebyschev's inequality, which holds for any probability distribution of X_i, establishes that $Pr(|X_i - \mu_i| \leq k\sigma_i) \geq 1 - 1/k^2$ for any $k \geq 1$. We use this rule to bound the probability that a given frequency x of a phrase α belongs to group i. That is, we require that the bound tells that frequency x belongs to group i with probability at least p, for some p close to 1. By setting $k = 1/\sqrt{1-p}$, we have that $Pr(|x - \hat{\mu_i}| \leq \hat{\sigma_i}/\sqrt{1-p}) \geq p$. Therefore, only when the frequency of a phrase α becomes $x \geq \hat{\mu_i} - \hat{\sigma_i}/\sqrt{1-p}$, we will assume that α deserves belonging to group i.

Estimators $\hat{\mu_i}$ and $\hat{\sigma_i}$ are easily maintained when new phrases enter group i or when their frequencies increase. By using p values closer to 1, we are more conservative at the time of creating new phrases.

A possible problem with the compression method is that we could produce too many irrelevant phrases αW because α deserves a shorter codeword at the time αW is read, but phrase αW will not appear again. One possible way to address this is to ensure that the frequency of α significantly exceeds what is necessary to deserve a shorter codeword. In particular, we can use a different p_i for each group i. It makes sense to be more conservative for larger i, where more useless phrases are likely to be generated.

3.3 Searching and Decompressing PBDC Compressed Text

Using ETDC, exact search of a word implies just searching for it in the vocabulary, getting its codeword, and then seeking the codeword in the compressed

text using any Boyer-Moore family algorithm. However, using PBDC, a word can correspond to more that one slot of the hash table, because it can appear in one slot alone but there may also be slots storing phrases containing that word.

For this sake we maintain vector *codewordlist*. This is maintained only for slots that correspond to one-word phrases. For word X, the vector contains the list of codewords of all those phrases that include word X, for example phrases XY, YX, YXZ, etc. This list is easily updated during the compression process because each time a new phrase αW is added to the vocabulary, each of its words has just been individually read. Then, using the same hash function, the slots for all those words are efficiently found in order to update their *codewordlist* vector. The new codeword for αW is added to each of those *codewordlist*s. This could be easily extended to keep track of all phrases that contain each existing pair of words, and so on.

To search the compressed corpus for a word X, a trie structure is built from the *codewordlist* vector of the slot of X. Then we apply a Boyer-Moore family algorithm such as *Set Horspool* [7, 13]. More complex searches such as approximate or regular expression searching can be easily carried out by scanning the vocabulary and building the trie with all the *codewordlist*s of all the matching vocabulary words.

Decompressing a PBDC is a very efficient and straightforward process. Each codeword is easily parsed due to the flag bit of each byte marking the end of each codeword. Then the codeword is transformed to a position by the decode procedure ($i = decode(C_i)$). This position is used to index the *codewords array*, and then the slot where the encoded phrase is kept in the hash table is retrieved. Finally, the *phrases* pointers are used to retrieve the phrase words one by one (right to left). Notice that the fact that a word can be encoded with different codewords in different (composed) phrases does not affect this process at all.

4 Empirical Results

We used some large text collections from TREC-2, namely AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF), as well as from TREC-4, namely Congressional Record 1993 and Financial Times 1991 to 1994. We also concatenated them all, creating a corpus we called ALL, with more than one gigabyte and 885,630 different words. We used the spaceless word model [15] to create the vocabulary; that is, if a word was followed by a space, we just encoded the word, otherwise both the word and the separator were encoded.

Our first experiment has to do with the number of phrases generated depending on parameter p of Section 3.2. We used AP corpus, of 238 megabytes and 269,141 words. The two sets of results were obtained using 10% and 5% of AP for the first phase (ETDC), respectively. The number of phrases produced in groups 2, 3, and 4, depends on the probabilities p_i we use. Table 1 shows the results. Group 4 does not have concatenations because there are no phrases encoded with 4-byte codewords except in the least conservative case. Notice how the compression ratio improves as the number of concatenations grows when we

prob.	Phase 1 uses 10% of AP					Phase 1 uses 5% of AP				
	group 2	group 3	group 4	tot.	Ratio%	group 2	group 3	group 4	tot. conc	Ratio%
0.995	2	14,592	0	14,594	32.229	67	26,457	0	26,524	32.226
0.990	3,803	117,079	0	120,882	31.603	2,441	135,812	0	138,253	31.584
0.950	138,745	1,497,842	0	1,636,587	27.974	97,502	1,668,667	0	1,766,169	27.617
0.900	276,858	2,118,248	76,616	2,471,722	26.862	211,888	2,284,394	94,001	2,590,283	26.489

Table 1. Trade-off among compression ratio and number of concatenations.

are less conservative and use $p = 0.9$ instead of $p = 0.99$. Of course this must be weighted against the larger vocabulary of phrases we must store.

We focus now on comparing PBDC against alternative approaches, where small portions of the ALL corpus are used to initialize the model and then all the rest of the corpus is added. In our case, we compress the first small part with ETDC and the rest with PBDC. We test the latter with two groups of p_i parameters: $p_2 = 0.9$, $p_3 = 0.99$ and $p_4 = 0.999$, and the more conservative $p_2 = 0.99$, $p_3 = 0.999$ and $p_4 = 0.9999$.

We also use ETDC in a *No-Concatenating mode*. This is exactly what is proposed in previous work [10, 11] to handle growing collections, just using ETDC instead of Huffman. That is, during the second phase, new words are added to the vocabulary, but changes in the word frequency distribution are not considered.

Table 2 presents the results. Each row shows the portion (1%, 5% and 10%) of the corpus compressed during the first phase. The last row is a special case. It shows the resulting data when the semi-static ETDC approach was used over the whole corpus.

Columns 2 and 3 give the size in kilobytes and the number of words of the initial text. The fourth column shows the compression ratio achieved with ETDC using the *No-Concatenating mode* (previous work). Columns 5 to 7 and 8 to 10 show compression ratio, number of concatenations and number of phrases, respectively, with the two settings for p_i. The number of phrases is that of concatenations plus the number of single words. This latter number can be smaller than the vocabulary size because in PBDC some words may appear only as part of phrases.

The table shows that the compression ratio is always better using PBDC. We must notice immediately, however, that this result is misleading as it is not considering the size of the vocabulary, which is much larger with PBDC. Although vocabularies are usually kept in main memory, a version of them (maybe compressed in some form) must be stored in secondary memory and accounted for in the final space requirement of the method.

A possible storage method for vocabularies is as follows. The *No-Concatenating* mode using ETDC only needs to store a vocabulary array where words must be

phase 1			No conct.	Phs 2 (0.9;0.99;0.999)			Phs 2 (0.99;0.999;0.9999)		
% ALL	K bytes	#vocab	ratio	PBDC %	#concat	#phrases	PBDC %	#concat	#phrases
1%	10,807	67,559	34.831%	28.577%	2,916,577	3,752,456	30.957%	941,131	1,803,325
5%	54,036	130,585	34.636%	28.949%	2,844,572	3,684,929	31.033%	916,273	1,782,660
10%	108,072	178,050	34.607%	29.117%	2,717,952	3,558,747	31.020%	878,028	1,740,197
100%	1,080,720	885,630	32.877%	32.877%	0	885,630	32.877%	0	885,630

Table 2. Compression of ALL Corpus (1,080,720,304 bytes). We do not count the vocabulary sizes.

phase 1			No conct.	Phs 2 (0.9;0.99;0.999)			Phs 2 (0.99;0.999;0.9999)		
% ALL	K bytes	#vocab	ratio	PBDC %	#concat	#phrases	PBDC %	#concat	#phrases
1%	10,807	67,559	35.611%	31.246%	2,916,577	3,752,456	32.346%	941,131	1,803,325
5%	54,036	130,585	35.416%	31.571%	2,844,572	3,684,929	32.406%	916,273	1,782,660
10%	108,072	178,050	35.387%	31.657%	2,717,952	3,558,747	32.369%	878,028	1,740,197
100%	1,080,720	885,630	33.657%	33.657%	0	885,630	33.657%	0	885,630

Table 3. Compression of ALL Corpus (1,080,720,304 bytes). We count the size of the vocabularies.

sorted in codeword order. The vocabulary needs 8,428,001 bytes (a 0.780% of the text). In the PBDC approach, an array with as many entries as phrases must be stored. The entries must be sorted in codeword order. The first byte of each entry is used to encode whether the entry is a single word or a multi-word phrase. If it is a word entry, the byte will give its length, but if it is a phrase entry that first byte will be 0. In the phrase type, after this first byte, the entry has 2 pointers using 3 bytes each, corresponding to the *phrases* entries. Therefore an overhead of 7 bytes for each concatenation must be paid for the PBDC approach in addition to the vocabulary of the ETDC.

The space needed to store frequency information can be reduced by using an appropriate compressor, as most phrases with long codewords will share low frequencies. Just applying classic Huffman entails an overhead of less than 1 byte per phrase. A more sophisticated approach, encoding frequencies (as if they were word ranks) with ETDC and then applying classic Huffman to the output bytes reduces the overhead to 0.25 bytes per phrase.

If the vocabulary is stored with the compressed text and we want to restart the system for the digital library, all the structures shown in Figure 1 will need to be rebuilt. Notice that the codeword list of each slot can be easily obtained because each phrase is linked with all the former ones in a way that all the single words can be reached.

Table 3 shows the same results, now considering the vocabulary sizes. It can be seen that PBDC significantly outperforms previous approaches (the *No-Concatenating* mode) by around 12%. It can also be seen that the least conservative approach works better even when it has to pay for the larger vocabulary generated. Finally, we note that PBDC works *better* when it uses a smaller initial ETDC phase, which suggests that it could be used as a replacement of ETDC in general, not only with the aim of updating the text collection.

5 Conclusions and future work

We have presented *Phrase Based Dense Code (PBDC)*, a new compression method that is useful for digital libraries because it reaches good compression ratios that do not degrade when the text collection grows. On the other hand, PBDC allows direct search over the text and efficient decompression of arbitrary portions of the collection.

PBDC is a hybrid method because it takes advantage not only of the word frequency distribution but also of the co-occurrence of words. Although more tuning and experiments have to be performed in real digital library scenarios, and better implementations of PBDC have to be developed, the promising empirical results show that this hybrid approach can lead to new compression methods.

We emphasize that the idea of inserting phrases as source symbols of a statistical encoder has independent interest and can be used in broader scenarios.

It is clear that more experimentation is necessary to optimize the probability parameters that obtain the best compression keeping a relatively low number of concatenations, and more research must be done to have better criteria about when a new concatenation is going to be useful. One possibility is to look for the probability of the following word, because if that is not a common word, the new phrase will probably not appear ever again. It would also be interesting to explore the use of linguistic techniques to choose phrases with higher probability of future occurrence.

References

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. AW, 1999.
2. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. P.Hall, 1990.
3. N. Brisaboa, A. Fariña, G. Navarro, and José Paramá. Simple, fast, and efficient natural language adaptive compression. In *Proceedings of the 11th SPIRE*, LNCS 3246, pages 230–241, 2004.
4. N.R. Brisaboa, E.L. Iglesias, G. Navarro, and J.R. Paramá. An efficient compression code for text databases. In *25th ECIR*, LNCS 2633, pages 468–481, 2003.
5. E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM TOIS*, 18(2):113–139, 2000.
6. H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Acad. Press, 1978.
7. R. N. Horspool. Practical fast searching in strings. *SPE*, 10(6):501–506, 1980.
8. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, 1952.
9. A. Moffat. Word-based text compression. *SPE*, 19(2):185–198, 1989.
10. A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. *KDE*, 9(2):302–313, 1997.
11. E. Moura. *Compressao de Dados Aplicada a Sistemas de Recuperacao de Informacao*. PhD thesis, Universidade Federal de Minas Gerais, Brazil, 1999.
12. G. Navarro, E.S. de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *IR*, 3(1):49–77, 2000.
13. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. CUP, 2002.
14. G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. 11th CPM*, LNCS 1848, pages 166–180, 2000.
15. E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. 21st SIGIR*, pages 298–306, 1998.
16. I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kauffman, 1999.
17. G.K. Zipf. *Human Behavior and the Principle of Least Effort*. AW, 1949.
18. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE TIT*, 23(3):337–343, 1977.
19. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE TIT*, 24(5):530–536, 1978.