Improving semistatic compression via phrase-based modeling[☆]

Nieves R. Brisaboa[a], Antonio Fariña[a], Gonzalo Navarro[b], José R. Paramá[a,*]

[a] *Database Lab, Facultade de Informática, University of A Coruña, Campus de Elviña s/n, 15071 A Coruña, Spain*

[b] *Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile*

Abstract

In recent years, new semistatic word-based byte-oriented text compressors, such as Tagged Huffman and those based on Dense Codes, have shown that it is possible to perform fast direct search over compressed text and decompression of arbitrary text passages over collections reduced to around 30-35% of their original size. Much of their success is due to the use of words as source symbols and a byte-oriented target alphabet. This approach broke with traditional statistical compressors, which use characters as source symbols and a bit-oriented target alphabet.

In this work we go one step beyond by using phrases as source symbols. We present two new semistatic modelers that we combined with a dense coding scheme to obtain two new compressors: *Pair-Based End-Tagged Dense Code (PETDC)*, where source symbols can be either words or pairs of words, and *Phrase-Based End-Tagged Dense Code (PhETDC)*, which considers words and sequences of words (phrases). PETDC compresses English texts to 28-29% and PhETDC to around 23%, outperforming the optimal byte-oriented zero-order prefix-free word-based semistatic compressor by up to 8 percentage points. Moreover, PETDC and PhETDC still permit random access and efficient direct searches using fast Boyer-Moore algorithms.

Keywords: Text compression, Direct search.

[☆] A preliminary version of this paper was published in (Brisaboa et al., 2006).

[*] Corresponding author. Tel. +34981167000 Fax. +34981167160.

Email addresses: brisaboa@udc.es (Nieves R. Brisaboa), fari@udc.es (Antonio Fariña), gnavarro@dcc.uchile.cl (Gonzalo Navarro), parama@udc.es (José R. Paramá)

1. Introduction

A lossless compressor is evaluated by the amount of space reduction it achieves, as well as by its compression and decompression efficiency. Decompression is usually more important since in many cases the file is compressed once and decompressed many times. When using a compressor within the framework of a text database, however, more factors come into the evaluation. In such a case the compressor is required to support two further operations: *i)* direct search into the compressed text without decompressing it, and *ii)* local decompression (random access) of any portion of the compressed file without the need of decompressing it from the beginning.

Statistical compressors replace the most frequent source symbols by shorter codewords. Therefore, they need a *model* of the original file which informs about the frequency of each source symbol. There are three strategies to obtain such a model: static, semistatic, and dynamic. In this work, we deal with semistatic modelers. Compressors using a semistatic modeler perform a first pass over the original text to obtain the list of distinct source symbols and to count their number of occurrences. Once the model is built, an *encoding scheme* assigns each source symbol a codeword. Then, the compressor performs a second pass over the original file replacing each source symbol by the corresponding codeword. Finally, to inform the decompressor of the correspondence between codewords and source symbols, a prelude is stored along with the compressed text.

Classical statistical compressors are not well suited for text databases since they use character-based modelers, which even using a Huffman coding (Huffman, 1952) obtain poor compression ratios[1] (around 60%).

Bentley, Sleator, Tarjan, and Wei (1986) used words as source symbols instead. This might seem problematic, as the size of the prelude increases significantly. Yet, since the number of different words grows sublinearly with the text size (Heaps, 1978), and the distribution of words is much more biased than that of characters (Zipf, 1949), compression improves appreciably if the text is large enough (Moura, Navarro, Ziviani, and Baeza-Yates, 2000). For example, a Huffman-based compressor using a word-based modeler can reach compression ratios around 25% (Moffat, 1989).

[1] Compression ratio is the size of the compressed text as a percentage of its original size.

Byte-oriented variants of Huffman codes allow for faster decompression in exchange for higher compression ratios. These stay around 30% for the basic Plain Huffman, and 35% for the so-called Tagged Huffman codes (Moura et al., 2000). The latter, aimed at allowing fast direct searches on the compressed text, have been superseded by the more recent *Dense Codes* family (Brisaboa, Fariña, Navarro, and Paramá, 2007). For example, the End-Tagged Dense Code (ETDC) is a statistical semistatic compressor using a semistatic zero-order word-based modeler.[2] It retains the fast direct searchability of Tagged Huffman codes, permits local decompression in both directions from a point in the text, reduces compression ratios to around 31%, and is simpler to program.

One way to improve compression ratios beyond the 30% achievable with byte-oriented compression is to use higher-order modeling, that is, to capture the dependencies between consecutive words in the text. For example, in a newspaper article it is feasible to find a considerable number of sequences of words like "European Union" or "United States". This was indeed mentioned in the foundational article of Bentley, Sleator, Tarjan, and Wei (1986), but not explored further.

The idea of capturing the co-occurrence of sequences is behind high-order statistical compressors and repetition-based compressors (Bell, Cleary, and Witten, 1990). One general problem in those approaches is that it becomes difficult to perform random access to the text, as well as to carry out direct pattern matching on it. In addition, it is not obvious how to spot the text repetitions in the best way. A well-known one-pass heuristic is the LZ77 compressor (Ziv and Lempel, 1977). This performs well in practice, but it is particularly unfriendly to random access and direct searches.

Contrarily, *offline* compressors (Apostolico and Lonardi, 2000; Katajainen and Raita, 1989; Larsson and Moffat, 1999; Rubin, 1976; Turpin and Smyth, 2002) spend a good deal of time and/or space during compression, since the idea is to compress static databases to be stored in CD or DVDs. Some afford several passes over the original text and/or a high memory consumption to achieve high compression, yet decompression is fast and memory-efficient.

These compressors use a *phrase book* and encode the input as a list of pointers to

[2]Zero-order modelers provide the probability of each source symbol without taking into account the surrounding symbols in the source stream.

entries in the phrase book (Rubin, 1976). By using an algorithm for finding the shortest path in a network, Katajainen and Raita (1989) gave a procedure that, given a phrase book, obtains a time-efficient approximation algorithm for the space-optimal encoding. Re-Pair (Larsson and Moffat, 1999) performs several passes over the text replacing pairs of adjacent source symbols by new symbols until all pairs occur only once. Offline (Apostolico and Lonardi, 2000) calculates a measure of compression gain for all the possible non-overlapping substrings of the input string; those strings with high gain are selected as a phrase for the phrase book. Crush (Turpin and Smyth, 2002) uses Crochemore's algorithm (Crochemore, 1981) to create a list of phrases that are used to encode the text as the compressor traverses the original text.

In this paper we improve the compression ratio of ETDC by coupling it with an offline modeler that detects promising sequences of words and regards them as a single token. As a result, compression ratios become competitive with the best compressors (near 25%). Compression speed is reasonable, while decompression, random access, and direct search speeds stay very appealing for compressed text database scenarios.

More precisely, we introduce two new semistatic modelers that, coupled with the encoding scheme of ETDC, produce two new compressors. The first one is called *Pair-Based End-Tagged Dense Code (PETDC)*, with a compression ratio around 28-29% for sufficient large texts. It builds a model of the text considering two types of source symbols: words and pairs of words. *Phrase-Based End-Tagged Dense Code (PhETDC)* goes one step further by building a model composed of words and sequences of words (phrases). Its compression ratio improves up to around 23%.

The key point is how the pairs and phrases are selected. For PETDC, we cannot add all the possible pairs because the size of the prelude would spoil the compression. We have devised a method to select the pairs of words that most improve the compression, taking into account the cost in space of adding such a pair to the prelude. To construct phrases, PhETDC uses the replacement algorithm of Re-Pair, which is preempted to speed up compression. As the resulting sequence of phrases is still compressible, PhETDC applies a dense coding scheme over it.

2. End-Tagged Dense Code

In general, ETDC can be defined over symbols of b bits, although in this paper we focus on the byte-oriented version where b=8. Given source symbols with decreasing probabilities $\{p_i\}0 \leq i < n$ 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 sequentially.

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 $\underline{0}0000000{:}\underline{1}0000000$, the 130^{th} as $\underline{0}0000000{:}\underline{1}0000001$ and so on until the $(128^2{+}128)^{th}$ word which is encoded as $\underline{0}1111111{:}\underline{1}1111111$. Note that the code depends on the rank of the words, not on their actual frequency. As a result, the prelude only includes the sorted vocabulary (list of source symbols).

However, not only is the sequential procedure available to assign codewords to the words. There are simple *encode* and *decode* procedures that can be efficiently implemented, because the codeword corresponding to the 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 $k = \left\lfloor \frac{\log_2\left(2^{b-1} + (2^{b-1}-1)i\right)}{b-1} \right\rfloor$, and adding 2^{b-1} to the last digit.

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

A deeper description of ETDC, including analytical and empirical studies, can be found in Brisaboa et al. (2007).

3. Pair-Based End-Tagged Dense Code

PETDC, as all semistatic compressors, performs a first pass over the original text in order to gather the vocabulary and the number of occurrences of each source symbol. However, in addition to the initial vocabulary of words, PETDC also collects all the different pairs of words that occur adjacent in the source text and counts their number of

occurrences. PETDC aims at taking advantage of the co-occurrence of words in the text by including some pairs in the vocabulary, thus comprising both single-words and pairs. Its main idea is simple: in classic semistatic compressors, each symbol in the vocabulary has a unique codeword assigned by the encoding scheme. Therefore, replacing two source words by just one codeword during the second pass may need fewer bytes than replacing two single words by two codewords.

Example 1. *Consider a text containing the following one-character words ADCBACDCCDABABACBB. Being only 4 source symbols ('A', 'B', 'C', and 'D'), the ETDC encoding scheme assigns codewords of only 1 byte to all of them. As a result, the size of the compressed file (compressed text plus the prelude) is 18+4=22 bytes.*

Let us add the most frequent pair of words ('BA') to the vocabulary and compress the same text again. In this case, the vocabulary contains the symbols: 'C', 'D', 'BA', 'A', and 'B'. Now the compressed text occupies 15 bytes and the prelude needs 6 bytes (the new pair is added as two pointers to the positions of the plain words forming the pair). Therefore, the compressed file requires 15+6=21 bytes.

3.1. Deciding which pairs should be added to the vocabulary

Adding all the different pairs to the vocabulary does not usually improve the compression ratio because the prelude would grow too much. Figure 1(a) shows the evolution of the size of a compressed file (as the sum of the size of the compressed text and the size of the prelude) depending on the number of pairs added. The process starts adding the most frequent pairs and continues adding pairs by decreasing order of frequency. As expected, the first pairs improve the compression effectiveness greatly. However, at some point, the gain obtained by replacing two words by a unique codeword does not compensate the growth of the prelude.

Figure 1(b) shows that the curve has multiple local minima. This fact prevents us from breaking the addition of pairs to the vocabulary when the addition of a new pair worsens the compression. Instead of that, as in Apostolico and Lonardi (2000), PETDC uses a gain function that determines which ones have to be added.

3.1.1. Gain function

Let us assume that a pair $\alpha\beta$, composed of two words α and β, is a candidate to be added to the vocabulary. Let us define f_x as the number of occurrences of a word or pair x. Let us also define C_x as the codeword assigned to x by the encoding scheme, and let $|C_x|$ be the length of that codeword. The gain function is based on comparing the number of bytes needed to encode all the occurrences of α and β in two cases:

1. The pair is skipped ($skip_{bytes}$).
2. The pair is added to the vocabulary (add_{bytes}).

Once those values are computed, the pair $\alpha\beta$ is added to the vocabulary if $skip_{bytes} > add_{bytes}$ and skipped otherwise. Values $skip_{bytes}$ and add_{bytes} are given by the two following expressions:

$$skip_{bytes} \leftarrow f_\alpha * |C_\alpha| + f_\beta * |C_\beta| \text{ and}$$
$$add_{bytes} \leftarrow f_{\alpha\beta} * |C_{\alpha\beta}| + (f_\alpha - f_{\alpha\beta}) * |C'_\alpha| + (f_\beta - f_{\alpha\beta}) * |C'_\beta| + K,$$

where C'_α and C'_β are the codewords assigned to the words α and β, assuming that the pair $\alpha\beta$ is added and therefore their number of occurrences are $f_\alpha - f_{\alpha\beta}$ and $f_\beta - f_{\alpha\beta}$, respectively. The term K is an estimation of the number of bytes needed to store any pair into the vocabulary. In our implementation, K=5 bytes.

After adding a pair $\alpha\beta$ to the vocabulary, it is necessary to ensure that any pair ending in α or beginning in β will not be included later. This happens because, once we choose a pair, we do not access the text to replace it by another symbol in a Re-Pair-like fashion. This makes compression faster, but some pairs must be discarded. As a result, given the text $'\gamma\underline{\alpha\beta}\delta\alpha\mu'$, the addition of the pair $\alpha\beta$ implies that the pairs $\gamma\alpha, \beta\delta$, and $\delta\alpha$ can no longer be added to the vocabulary. This is done just by marking the word α as "disabled as last word of pair", the word β as "disabled as first word of pair", and finally checking those flags before adding a new pair to the vocabulary.

3.2. Data Structures

The data structures used by the compressor are sketched in Figure 2. There are two well-defined parts:

1. Data structures that make up the vocabulary.

2. Data structures needed to hold the candidate pairs.

The vocabulary of the compressor consists of: a hash table used to locate a word or pair quickly (*hashSymb*) and two vectors: *symbVect* and *topVect*. The hash table *hashSymb* contains eight fields:

1. *type* indicates if an entry is either a word w or a pair p,
2. *word* stores the original word in plain text form, if *type* is set to w,
3. *freq* keeps the number of occurrences of the entry,
4. e_1 and e_2 flag if the word is enabled to be the first or second component of a pair, respectively,
5. w_1 and w_2 store, for an entry of type p, pointers to the words that form the pair, and
6. *code* stores the codeword assigned to each entry of the vocabulary after the code generation phase. This field could actually be replaced by calls to the function *encode(i)*.

The vector *symbVect* maintains the vocabulary sorted by frequency. Its first slot points to the entry of *hashSymb* where the most frequent word (or pair) in the source text is stored, the second slot points to the second most frequent source symbol, and so on. Assuming that *symbVect* is sorted by decreasing frequency, $topVect[f_i]$ keeps track of the first entry in *symbVect* whose frequency is f_i. If there are no symbols of frequency f_i, then $topVect[f_i]$ points to the position of the first symbol j in the *ascending* sort of the vocabulary such that $f_i < f_j$.

The offline compressors that use a gain function need to calculate the frequency of the phrases and an estimation of the space consumed by their compressed version. This may result in a bottleneck that our compressors should alleviate in order to obtain reasonable compression times. Recall that in ETDC, we do not need the actual frequency of the source symbols to compute their codewords, we only need their position in the ordered vocabulary. Maintaining the vocabulary ordered upon insertions and deletions would be too expensive. Therefore, to estimate the positions of the source symbols, we only maintain *topVect* updated. Then, being x an entry with frequency f_x, we estimate $|C_x|$

as $|C_{(topVect[f_x])}|$. Although it gives an approximate value, computing this estimation is much cheaper than computing the actual codeword sizes.

Finally, managing the candidate pairs to be added to the vocabulary includes the use of two auxiliary data structures:

1. A hash table *hashPairs,* with fields *freq*, w_1, and w_2, used to give a fast access to each candidate pair.
2. A vector *pairsVector*, which maintains all the candidate pairs sorted, in the same way as *symbVect* does.

3.3. Compression, Decompression, and Search Procedures

Compression consists of five main phases:

1. *First pass along the text.* As shown, during this pass, PETDC obtains the different v original words and the different p candidate pairs that appear in the text. At the same time, the number of occurrences of all those elements is obtained. The process costs $O(n)$, being n the number of words in the text. When the first pass ends, the vectors *pairsVect* and *symbolVect* are sorted by decreasing frequency. Finally, *topVect* is initialized: consider m the maximum frequency value of the v original words, then for all i=m downto *1*, we set *topVect[i]*←j, if j is the first entry in *symbVect* such that *hashSymb[symbVect[j]].freq*=i. If $\nexists j$ such that *hashSymb[symbVect[j]].freq*=i, then *topVect[i]*←*topVect[i+1]*.
   The overall cost of this first phase is $O(n+v\log v+p\log p+m)$. We show empirical evidences that $n \gg p$ and $n \gg m$, and typically $p > v$; in which case the cost is $O(n)$. Figure 2(a) shows the state of the structures after this phase.
2. *Choosing and adding candidate pairs.* During this phase, *pairsVector* is traversed $O(p)$ time. A candidate pair $\alpha\beta$ is either added to the vocabulary or discarded by applying the *gain function* explained in Section 3.1.1. To compute the gain function, as seen, using *topVect* we compute the current positions[3] of α and β in the vocabulary. We also need to assume that the pair $\alpha\beta$ is added to the vocabulary and to compute the new ranks of $\alpha\beta$, α, and β in the new ordered vocabulary.

[3]Using the encoding scheme of ETDC, we can compute in $O(\log(i)/b)$ time the codeword C_i = *encode*(i), where i is the rank of a word w_i in the vocabulary.

The cost of keeping *topVect* sorted after the insertion of $\alpha\beta$ is $O(m)$ time, since this can be done with a single pass, by making simple additions and subtractions. Of course, $\alpha\beta$ is also inserted into *hashSymb* and into *symbVect*. Observe in Figure 2(b) the state of the structures after adding the pair 'BA' to the vocabulary. Since the pair 'BA' (more specifically, its position in *hashSymb*) is placed in the fifth position of *symbVect,* this vector no longer contains the vocabulary sorted: 'BA' has 3 occurrences and therefore it is the fourth most frequent element of the vocabulary. However, observe that *topVect* is updated since, for example, the seventh entry is now empty due to lack of vocabulary entries with frequency 7. Now, *topVect* contains the pointers to a hypothetic updated version of *symbVect*, which no longer exists.

The overall cost of this phase is $O(p_a m + p) = O(p_a m)$, being p_a the number of pairs added to the vocabulary.

3. *Code Generation Phase.* The only data structures needed in this phase are depicted in Figure 2(c). The vocabulary (with v' entries) is ordered by frequency and the encoding scheme of ETDC is used. Encoding takes $O(v')$ time. As a result, *hashSymb* will contain the mapping $entry_i \rightarrow code_i \ \forall i \in 1 \ldots v'$. The cost of this phase is $O(v' \log v')$.
4. *Second pass.* The text is again traversed reading two words at a time and the source words are replaced by codewords. If the read pair $\alpha\beta$ is in *hashSymb*, then the codeword $C_{\alpha\beta}$ is output and two new words $\gamma\delta$ are input. Otherwise C_α is output and only the following word γ is read to form a new pair $\beta\gamma$. This phase takes $O(n)$ time.
5. *Storing the prelude.* As in ETDC, the prelude is stored along with the compressed data to permit decompression. A bitmask is used to save the type of entry. Then the v' entries of the vocabulary follow that bitmask. A normal entry stores plain text, but if it stores a pair $\alpha\beta$, the entry stores the relative positions of α and β in the vocabulary (encoded with the on-the-fly $C_i \leftarrow encode(i)$ function in order to save space). Finally, the whole prelude is encoded with character-based Huffman.

Considering the cost of each phase, the overall cost of the whole process is $O(n + p_a m + v' \log v' + n)$. Given that $v' \ll n$, we obtain that the overall cost is $\mathrm{O}(n + p_a m)$.

Note that since p_a is O(n), time is quadratic at most. However, in practice $p_a \ll n$.

Decompression starts by loading the prelude into a vector that keeps both words and pairs. For each codeword C_i in the compressed file, $i \leftarrow decode(C_i)$ is used to obtain the entry i containing either the word or the pair associated to C_i.

When searching text compressed with PETDC, a searched word α can occur alone or as a part of one or more pairs $\alpha\beta$, $\gamma\alpha, \ldots$ Therefore, we have to use a multi-pattern matching algorithm. When we load the vocabulary, for each single-word α, we can easily generate a list with all the codewords that represent any pair including such a word α. After that, an algorithm from the Boyer-Moore family such as *Set Horspool* (Horspool, 1980; Navarro and Raffinot, 2002) is used to search for both all those codewords and codeword C_α.

4. Phrase-Based End-Tagged Dense Code

PhETDC goes one step beyond with regard to PETDC, since it represents phrases, with an undetermined number of words, using just one codeword.

However, the prelude issue gets more complicated because the number of possible entries in it is much higher. Therefore, the main problem is again how to determine which phrases should be introduced in the vocabulary to improve the compression ratio.

PhETDC is an evolution of PETDC based on the Re-Pair compression algorithm (Larsson and Moffat, 1999). PhETDC performs a first pass over the original text to obtain both the list of plain words and all the pairs of adjacent words present in the text. In this pass, the frequency of all those elements is collected and the original text is replaced by a vector of integers, where each occurrence of a word is substituted by the integer assigned to it. This integer is a pointer to the entry storing the replaced word in the hash table *hashSymb*.

Again, single words are immediately added to the vocabulary, but the pairs of words are kept in a list of candidate pairs, which may be added later to the vocabulary.

The second phase of the compression process begins by adding the most frequent pair $\alpha\beta$ of the list of candidate pairs to the vocabulary. Next, in the sequence of integers representing the text, all the occurrences of the selected pair are substituted by the new integer pointing to the entry in *hashSymb* that stores the new pair. This produces a

reduction in the number of occurrences of some pairs, and even some of them might disappear. For each occurrence of the new pair $\alpha\beta$, the affected pairs are:

1. The pair $\gamma_i\alpha$ formed by the previous word (γ_i) and the first word of the added pair (α).
2. The pair $\beta\delta_i$ formed by the second word of the added pair (β) and the following word (δ_i).

In addition, new candidate pairs appear. For each occurrence of the added pair ($\alpha\beta \to \lambda$), new candidate pairs/phrases appear. They are formed by:

1. The previous word and the added pair ($\gamma_i\lambda$).
2. The new pair and the next word ($\lambda\delta_i$).

Example 2. *Let us consider the text CDBCDCABCDBCACD. The most frequent pair is CD with 4 occurrences, and the second one is BC with 3. Then, we add the new element CD to the vocabulary and we replace all the occurrences of CD by the symbol E. The new text is* ***E****B****E****CAB****E****BCA****E****.*

Now, the pair BC has only 1 occurrence, whereas two new pairs (BE, EC) appear. As explained, BE and EC are pairs that are indeed phrases, since the symbol E is a representation of the pair CD. Therefore, the pair BE is actually a representation of the text BCD.

Once this replacement is finished, the processed pair is removed from the list of candidate pairs and the process continues. The replacing process continues by choosing the most frequent pair from the list of candidate pairs. Observe that, from that point on, two words, two pairs, or one word and a pair can form pairs.

Once the replacement process is finished, we use the ETDC encoding scheme to encode the resulting vector of integers. The prelude is similar to that of PETDC, but now the pointers of an entry can point to other pairs (while in PETDC, they only point to original words). Figure 3 shows the main steps of the compression process.

4.1. Deciding which phrases should be added to the vocabulary

Figure 4 shows how the compression process, explained in the previous section, evolves as we add all the candidate pairs in decreasing order of frequency to the vocabulary.

Again, the compressed file gets shorter up to a point where it starts to grow (in this experiment, this point was reached after the addition of 6,550 pairs and when the file occupied 816,757 bytes). We studied two methods to avoid adding pairs that worsen the compression: using a finishing condition and using a gain function.

Stopping the addition of phrases when their number of occurrences was under a given threshold value worked well. However, we obtained better results when that finishing condition was removed and we processed all the candidate pairs, choosing, by means of a gain function, those pairs that improved the compression.

In addition, we also studied if choosing the most frequent pair yields the best ordering to add pairs. Several strategies were tested, each one with different motivations:

1. The first strategy preselects the pair that, once added to the vocabulary, produces the largest increase of the entropy per target symbol. This pair is finally chosen only if its addition reduces the compressed file in at least $K = 5$ bytes. The process ends when there is no pair in the list of candidates increasing the entropy. The purpose in this approach is to produce symbols carrying as much information as possible in order to reduce the number of symbols in the output.
2. The second strategy preselects the pair that, once added to the vocabulary, minimizes the value (*entropy per target symbol*) $\times$ (*number of symbols in the compressed file*). That is, it chooses the pair that minimizes the lower bound of the size of the compressed file. It only adds pairs that reduce the compressed file in at least $K = 5$ bytes, and stops adding new pairs when the list of candidate pairs becomes empty.
3. The last strategy preselects the most frequent pair from the list. As in the previous cases, a candidate pair is added to the vocabulary provided that such an addition reduces the compressed file by at least $K = 5$ bytes; otherwise it is discarded. The process finishes when the list of candidate pairs becomes empty.

Table 1 shows the results of our study. In addition to obtaining the best compression ratio, the third alternative (to preselect the most frequent pair from the list) is the fastest one (by far) in compression time. Therefore, this is the compression strategy used in the empirical results.

We note that the chosen strategy is very similar to Re-Pair on words (Wan, 2003). The main differences are (1) that we preempt the recursive pairing sooner while they

continue until every pair is unique; (2) that they compress the prelude while we store it in plain form; and (3) that they compress the final sequence with bit-oriented Huffman while we use ETDC. Those differences are important with respect to random access (as we can uncompress an area without having to decompress the prelude, which is usually a significant part of the output) and direct search (as ETDC can be efficiently searched, while bit-oriented Huffman cannot).

4.2. Data Structures

PhETDC uses the same structures of PETDC, with some modifications in the hash table *hashSymb*. In addition, it uses a new vector of integers (T_{ids}), which follows the ideas in Wan (2003). T_{ids} keeps a representation of the text where, as explained, words and pairs are replaced by an integer (*id*) indicating the position of the word/pair in the hash table *hashSymb*. When a pair is added, the compressor has to replace in T_{ids} all the occurrences of the two symbols forming the new pair (when they are adjacent) by the *id* of the new pair. In order to avoid the traversal of the whole text, the following structures are needed:

1. For each code (word/pair) there is a doubly linked list connecting all its occurrences. Then, during the replacement of two *ids* by a new *id* (pair), the compressor only has to follow the linked list of one of the replaced *ids* and check, for each occurrence, the presence of the other *id*.
2. Two pointers to the previous word/pair and the next word/pair in the text. With the replacement of two consecutive *ids* by just one *id*, one of the entries in *Tids* remains unused. To jump over these unused slots, each entry has these two pointers: one points to the previous valid entry and the other to the next valid slot.

The hash table *hashSymb* has the same fields as in the case of PETDC, except the e_1 and e_2 fields, since now PhETDC does not invalidate possible pairs with the addition of one pair. Furthermore, each entry has a new pointer to one of the occurrences of its word/pair in T_{ids}. In this way, we have access to the linked list of occurrences of such a word/pair.

Observe that now the pointers to the two members of a pair/phrase (w_1 and w_2) can point to plain words or to other pairs, unlike PETDC, where the two pointers of a pair

pointed to plain words.

Figure 5 sketches the structures of the compressor in a point of the compression procedure. The structures are shown after the addition of the phrase "the room". The hash table *hashPairs* includes only the fields *freq*, p_1 and p_2, the column with the text version of the pair/phrase (with grey stripes) is included only for illustration purposes.

The new phrase (*the room*) is inserted in the 5^{th} entry of *hashSymb*. Then, when the *ids* 3 and 6 (representing the plain words *the* and *room*) are adjacent in *Tids*, they are replaced by the *id* 5. After that replacement, in all the occurrences of *the room*, the *next word/pair* pointer skips the next slot (formerly used by the *id* 6, that is, by the word *room*).

4.3. Compression, Decompression and Search Procedures

Compression consists of six main phases:

1. *First pass along the text.* This phase is the same as in the case of PETDC. Recall that it costs $O(n)$.
2. *Substitution of the plain words by an id.* During this phase, the vector *Tids* is built. As shown, each word in the original text is represented by an *id* that refers to the position of such a word in *hashSymb*. We described this phase separately just for illustration purposes, but this phase can be done in conjunction with the previous one with no additional analytical cost.
3. *Choosing and adding candidate pairs.* During this phase, *pairsVector* is traversed $O(p)$ time . A candidate pair $\alpha\beta$ is added to the vocabulary if its addition decreases the size of the compressed file in at least $K = 5$ bytes. As shown in the case of PETDC, the estimation of the new size of the compressed text costs $O(m)$ time, where m is the maximum frequency value of a word/pair.

   Once the new pair $\alpha\beta$ is added to *hashSymb*, we access the list of occurrences of α. We traverse this list and create the list of occurrences of $\alpha\beta$ by checking the occurrences of α that are followed by β. The average number of occurrences of α is n/v.

   In conjunction with the previous process, for each occurrence of $\alpha\beta$: $\gamma_i\alpha\beta\delta_j$, where γ_i is the previous word/pair and δ_j is the next word/pair:

(a) the number of occurrences in *hashSymb* of the pairs $\gamma_i\alpha$ and $\beta\ \delta_j$ are decreased by one unit,

(b) either the pairs $\gamma_i\alpha\beta$ and $\alpha\beta\delta_j$ are added (if they have not been added yet) or their frequency is increased by one unit, and

(c) the list of occurrences of α and β and their frequency in *hashSymb* are updated.

Finally, the vector *topVect* is sorted.

The overall cost of this phase is $O(p_a(m + n/v))$, being p_a the number of pairs added to the vocabulary.

4. *Code Generation Phase.* This phase is the same as in PETDC. The cost of this phase is $O(v' \log v')$.
5. *Second pass.* The *Tids* vector is traversed (using the pointers *next*). For each valid slot, the corresponding codeword is output. This phase takes $O(n)$ time.
6. *Storing the prelude.* It works as in the case of PETDC.

Considering the cost of each phase, the overall cost of the whole process is $O(n + p_a(m + n/v) + v' \log v' + n)$. Since each time a pair occurring x times is replaced we shorten *Tids* length by x elements ($n \leftarrow n - x$), the term $p_a(n/v)$ is bounded by $\mathrm{O}(n)$. As it also holds that $v' \ll n$, we obtain that the cost is $\mathrm{O}(n + p_a m)$, the same bound as PETDC.

Decompression starts by loading the prelude into a vocabulary vector. For each codeword C_i, the function $i \leftarrow decode(C_i)$ is used to obtain the entry i that contains either the word or the pair associated to C_i. Observe that now, a pair may contain one or two pointers to other pairs, therefore a recursive process should inspect those pairs until we obtain the plain words, which form a phrase.

As in the case of PETDC, a word being searched for can be found alone or as a part of one or more pairs. Therefore, we need to gather all the codewords corresponding to pairs that include a given searched pattern. This task is supported by keeping two lists for each word/pair δ:

- *contained_by*(δ) stores pointers to all the pairs which directly contain δ.
- *codewords*(δ) stores the codewords that should be sought in case of searching for δ.

Searches start by loading the prelude that keeps the vocabulary of words/phrases. For each pair λ formed by $\phi\theta$, a pointer to λ is added to *contained_by*(ϕ) and *contained_by*(θ). Then we have to create the *codewords* list for each searched pattern α. Let us consider that *contained_by*(α) includes pointers to ρ, τ, and σ. In such a case, the codewords C_α, C_ρ, C_τ, and C_σ are added to the list *codewords*(α). Yet, until now, this list only keeps the pairs that directly contain α, and so we need to include also those codewords representing any pair containing ρ, τ, and σ. Thus, we access the *contained_by* lists of ρ, τ, and σ and recursively follow the same procedure to fulfill the *codewords* lists of ρ, τ, and σ. After that, the codewords in *codewords*(ρ), *codewords*(τ), and *codewords*(σ) are also added to *codewords*(α). Finally, the compressed data is scanned and the codewords in *codewords*(α) are searched for.

5. Empirical Results

We used some large text collections from TREC-2,[4] namely AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF), as well as one from TREC-4: Congressional Record 1993 (CR). As a small corpus, we used the Calgary corpus.[5] To create the vocabulary, we opted for the spaceless word model (Moura et al., 1998); that is, if a word is followed by a space, we just encode the word, otherwise both the word and the separator are encoded.

Our machine is an Intel Core2Duo E6420@2.13Ghz, with 32KB+32KB L1 Cache, 4MB L2 Cache, and 4GB of DDR2-800 RAM. It runs Ubuntu 7.04 (kernel 2.6.20-15-generic). We compiled with gcc version 4.1.2 and the options -m32 -09.

5.1. Compression Ratio, Compression time, and Decompression time

We compared PETDC and PhETDC against two semi-static word-based byte-oriented compressors, namely Plain Huffman (PH) (Moura et al., 2000) and ETDC, and against a Re-Pair[6] compressor coupled with a bit-oriented Huffman.[7] We also included four

[4] `http://trec.nist.gov`.

[5] We used a subset of the Calgary collection that includes only the text files: book1-2, bib, news, and paper1-6. It is available at `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`.

[6] `http://www.cbrc.jp/~rwan/software/restore.html`.

[7] `http://cs.mu.oz.au/~alistair/mr_coder`.

well-known general purpose compressors: Gnu *gzip,*[8] a Ziv-Lempel-based compressor; Seward's *bzip2,* [9] a compressor based on the Burrows-Wheeler transform; *p7zip*[10] that is a LZMA compressor with a dictionary of up to 4 GB; and *ppmdj1*,[11] a compressor based on an *arithmetic encoder* coupled with a k-order modeler. We set *ppmdj1* compression options to *–o12 –m256 –r1* (using a 12-order modeler, up to 256MB of memory, and rebuilding the model -rather than discarding it- when memory is exhausted). These options provided the best compression in most files. The other two parameterizable compressors (*gzip* and *bzip2*) were run with default options, aiming at providing the best compression/speed trade-off.

In any case, compression times consider that the 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 the prelude.

Table 2 shows the compression ratios achieved by the compressors. PH is the optimal semistatic zero-order word-based byte-oriented statistical compressor, yet PETDC beats it by around 1-3 percentage points and PhETDC by around 4-8 percentage points.

As the word-based statistical methods, PETDC and PhETDC perform worse in small collections due to the need of storing a large prelude. Yet, if the text is large enough, they can even compete with a powerful compressor such as *p7zip*.

As expected, PETDC obtains better results than *gzip* except in the smallest collection. In the largest collection, PETDC compresses only 1.3 percentage points worse than *bzip2*, whereas in medium size collections, the gap is around 3-4 percentage points. Finally, PETDC obtains compression ratios around 6-12 percentage points worse than those of *p7zip*, around 7-12 percentage points worse than those of Re-Pair, and it is overcome by *ppmdj1* by around 12-16 percentage points.

PhETDC is still unable of beating *gzip* in the smallest collection. Yet, when the texts are large enough, PhETDC compresses between 10 to 15 percentage points better than

[8] `http://www.gnu.org.`
[9] `http://www.bzip.org.`
[10] `http://www.7-zip.org.`
[11] `http://www.compression.ru/ds/.`

gzip, it outperfoms *bzip2* by around 1-3 percentage points, and it is on a par with *p7zip*, which overcomes PhETDC by less than 1 percentage point.

With compression ratios around 3-7 and 5-13 percentage points better respectively, Re-Pair and *ppmdj1* remain as the only two compressors that clearly beat PhETDC in compression ratio. Yet, these techniques do not produce efficiently searchable outputs: the pattern may appear in many different forms along the text, and thus they essentially need to decompress in order to search. Even if we used the original word-based Re-Pair (Wan, 2003), which achieves similar compression ratios and where searching for a word is simpler, still the need to decompress the prelude and the slowness of processing bit-oriented Huffman codes hamper random access and direct search capabilities, as explained. Therefore, as we will see, the key reasons that make our compression ratios inferior to the classical word-based Re-Pair are the same for our high decompression and search efficiency.

We performed an additional experiment to check the compression ratios of Re-Pair using an ETDC encoder as a final compression step, instead of a bit oriented Huffman encoder. The values were the following: 35.15% compressing Calgary, 22.54% compressing CR, 22.31% compressing ZIFF, and 17.14% compressing AP. This is still better than PhETDC due to the compression of the prelude. Yet, as we will see, the highly sophisticated prelude compression makes classical Re-Pair (based on characters or words) much slower at decompression, particularly making infeasible direct searches on it.

Table 3 shows that PETDC and PhETDC pay the extra cost of managing pairs or phrases during compression. PETDC is around 3 times slower than ETDC and PH. However, it is still faster than *gzip*, *bzip2*, *p7zip*, and *ppmdj1*. PhETDC is between 4.5-5 times slower than ETDC and PH, around 1.5-2 times slower than *gzip*, and it is on a par with *bzip2*. Yet, with similar compression ratios, it is around 2.5-7 times faster than *p7zip*. The comparison against *ppmdj1* shows that PhETDC compresses less but it is around 2-6 times faster. Re-Pair, as an offline compressor, obtains remarkable compression ratios, but at the expense of poor compression times, in part due to a greedy use of memory. Yet, we have to take into account that we did not use the block-wise version of Re-Pair, which reduces the memory consumption in exchange for poorer compression ratios. We used the normal version of Re-Pair to have a fair comparison with our non-block-wise

PETDC and PhETDC.

In decompression (see Table 4), the extra-cost of PETDC consists only in processing the bitmask in the header of the vocabulary file and rebuilding the pairs from the pointers to single-words. Therefore, due to a shorter compressed input file, it is around 25% faster than PH and ETDC, and up to 65% faster than *gzip*. In the case of *p7zip*, and especially in the case of *bzip2* and *ppmdj1*, the gaps are considerable, being PETDC around 3 times faster than *p7zip*, between 3-7 times faster than *bzip2*, and between 13-75 times faster than *ppmdj1*.

In the case of PhETDC, the extra cost of the recursive rebuilding of the phrases implies a slowdown. Now, PhETDC is on a par with ETDC and PH, it is around 17-22% faster than *gzip*, except in the smallest collection, and between 1-3 times faster than *p7zip*. *bzip2* and *ppmdj1* are again the slowest techniques, being PhETDC between 3-6 times faster than *bzip2* and between 13-55 times faster than *ppmdj1*.

As expected, Re-Pair is not so time-consuming at decompression. Indeed, in the small collection Re-Pair performed similarly to PETDC and PhETDC, but in the rest of collections, it was around 2.5-4.5 times slower than PETDC and PhETDC.

We also compared the decompression performance of Re-Pair using an ETDC encoder instead of a bit oriented Huffman. Firstly, we compared the decompression of the sequence of integers generated by the Re-Pair process, and found that using ETDC is twice as fast as using bit-oriented Huffman. Then we compared the overall Re-Pair decompression time. In this case, using ETDC rather than Huffman reduces the overall decompression times by around 3-9%.

As a conclusion, if one is interested in compression ratio, PETDC and PhETDC can now compete with *bzip2* and *p7zip*, at the cost of losing some compression speed compared with the classical PH and ETDC. Yet, as we will see in the next section, PETDC and PhETDC are efficiently searchable, which is a remarkable feature.

5.2. Search Speed

We performed single and multi-pattern searches over corpus AP. To choose our search patterns, we followed the model in (Moura et al., 2000) where each vocabulary word is sought with uniform probability, and extracted patterns chosen at random over the vo-

cabulary of corpus AP. We classified the generated patterns depending on their frequency and length as shown below.

We carried out two different experiments. The first one compares searches over text compressed with PETDC and PhETDC versus searching the uncompressed text. This shows how much search time is gained or lost due to having the text in compressed form. The second kind of comparison is against ETDC. This shows the loss in search speed with respect to ETDC, due to the management of pairs and phrases.

To search text compressed with ETDC, we use our own implementations of *Horspool* and *Set-Horspool* algorithms[12] (Horspool, 1980; Navarro and Raffinot, 2002): *Horspool* for single-pattern searches and *Set-Horspool* for multi-pattern searches. In the case of PETDC and PhETDC, since a pattern can be represented by several codewords, we have to use the multi-pattern *Set-Horspool* even for single-pattern searches.

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*[13] software (Wu and Manber, 1992a,b). *Agrep* returns 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 harmful 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'* obtained by removing all the pattern occurrences from the original text (AP corpus). Then we ran *Agrep -s* over *T'* and we scaled the resulting times assuming that $|T'| = |AP|$. This essentially shows the same statistics and reflects more accurately the real search cost of *Agrep.*

To choose the search patterns, we considered the vocabulary of corpus AP, we extracted sets of patterns with K words of length L at random. We consider lengths L= *5* and *10,* and sets of K= *5, 25, 50* or *100* patterns. Figure 6 shows the average times of running the searchers over 10 sets for each combination of L and K.

PETDC single-pattern search (although it is actually a multi-pattern search) is around

[12] http://vios.dc.fi.udc.es/codes.

[13] ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z.

2.7 times slower than that of ETDC and around 2.1-2.6 times slower than *Horspool* single-pattern searches over plain text. Yet, in single-pattern searches, PETDC at least matches (if not clearly surpasses) the *Agrep* results, and in multi-pattern searches, PETDC is 1.1-2.8 times faster than the plain text searchers.

PhETDC is around 1.5-2.25 times slower than PETDC. With patterns of length 5, it is on a par with *Agrep* single pattern searches, but it is 4.8 times slower than *Horspool*. In multipattern searches under the same circumstances, PhETDC is 5%-74% faster than plain text searchers. However, when the patterns are long, it is 16%-45% slower.

As a summary, PETDC and PhETDC perform better in multipattern searches than in single pattern searches, whereas plain text searchers perform better when the patterns are long. Two factors determine this situation. The first one is that the searcher over compressed text has to traverse a much shorter file. The second factor is that plain text patterns are usually longer than compressed patterns, since compressed patterns are only between 1 and 3 characters long (the usual size of codewords). This favors the plain text searchers, especially in single-pattern searches, as longer patterns allow longer shifts. However, in any multi-pattern Boyer-Moore-type search, when both longer and shorter patterns are sought, the most efficient choice is to truncate all of them to the shortest length and verify them upon the occurrence of their truncated version. Therefore, the more the patterns are sought, the more the chances to cut down the search patterns, obtaining shorter shifts. This harms the advantage of the plain text searchers, particularly when the plain patterns are not very long.

An interesting question is how PETDC and PhETDC behave when the frequency of the searched words increases, since it is expected that the more frequent a word is, the more the codewords (from its pairs) that will represent such a word, and therefore, the more costly the search will be. Therefore, we included another experiment to determine how this factor affects the searches. Figure 7(a) and Figure 7(b) display the times needed to search respectively, for 1 pattern and 100 patterns of different frequencies. In both cases, patterns of 5 characters (the average length of English words) were used. As expected, PhETDC is the most affected searcher, while the others are quite insensitive to this factor. Still, PhETDC is always faster than *Agrep*, and in most multipattern searches, faster than *Horspool* over plain text.

6. Conclusions and Future Work

We have introduced two new semistatic modelers and tested them in two compressors called *Pair-Based End-Tagged Dense Code (PETDC)* and *Phrase-Based End-Tagged Dense Code (PhETDC)*. They take advantage of using pairs of words or phrases (exploiting the co-occurrence of words) to improve the compression obtained by word-based semistatic techniques such as PH or ETDC.

In essence, the goal is to use offline techniques to improve the ETDC compression ratio, which is its weakest point if we compare ETDC with PPM-based compressors, offline compressors, *p7zip*, and *bzip2*. At the cost of a slowdown in compression and search speed, PETDC gets closer to *bzip2* and *p7zip* compression ratios, and even for sufficient large texts, PhETDC beats *bzip2* and it is on a par with *p7zip*, which obtains non-efficiently searchable compressed text. The comparison against a powerful PPM-based compressor such as *ppmdj1* shows that the worse compression of PETDC and PhETDC is compensated by their faster compression and decompression processes. We showed also that PETDC and PhETDC cannot compete with Re-Pair in compression ratio, but they still obtain competitive values with remarkable better compression time, which is the main disadvantage of Re-Pair, and an efficiently searchable compressed text.

As explained, the improvement in compression ratio has a price. PETDC is around 3 times slower than ETDC in compression, yet it is 25% faster in decompression. In the case of PhETDC, it is 4.5 times slower than ETDC in compression and it is on par in decompression speed.

Figure 8 shows trade-offs between compression ratio and both compression and decompression times. We used the corpus CR, where Re-Pair can run without swapping in our machine. In this way the figure reflects the actual differences better.

As expected, the improvement in compression ratio implies also a slowdown in the search procedure with respect to ETDC. PETDC is between 2.1-2.6 times slower than plain text *Horspool* single-pattern searches. Yet, in multipattern searches, it is still between 1.11-2.8 times faster than searching the plain text. In the case of PhETDC, it is faster than plain text searchers in multipattern searches for average length words, but it is beaten in searches for long patterns. Figure 9 displays trade-offs between compression ratio and search time.

As a conclusion, there is an interesting trade-off between speed and compression ratio. Those who are more interested in speed can use the classical ETDC, which compresses to around 31%. With a reasonable slowdown, PETDC obtains 27-28% compression ratio. Finally, users most interested in compression ratio can opt for PhETDC, which achieves up to 23%, at the cost of a more significant loss of speed.

As semistatic compressors, PETDC and PhETDC have to keep in memory the vocabulary (that is, the model). In the case of the word-based compressors, this consumes a considerable amount of memory, which in the case of PETDC and PhETDC is even higher, since they do not only deal with words, but also with pairs of words or phrases. In addition, PETDC and PhETDC use structures that are more complex and PhETDC requires recursive processes. Table 5 shows the peak memory consumption of PETDC and PhETDC compared with ETDC and Re-Pair, which are compressors that also have to store a large model. From the point of view of a practitioner, on the one hand, we remark that the implementation of PETDC is quite similar to that of the semistatic ETDC. The main difference is the need of managing pairs, and marking those that are either still valid or have already been discarded. Table 5 reflects this point, as it shows that the executable file of the PETDC compressor is only 5% larger than that of ETDC. On the other hand, it can be seen that the implementation of PhETDC requires much more effort. We show that the executable file of the PhETDC compressor is around 36% and 29% larger than those of ETDC and PETDC respectively.

As a future work, to reduce the memory usage of our compressors, we intend to develop block-wise versions of PETDC and PhETDC following the ideas in Wan and Moffat (2007). Block-wise compressors compute the model of a chunk of the text and use it to compress such a chunk. For the next chunk of text, the model is computed again. By using small chunks, models are shorter and then the memory consumption decreases. As shown in Wan and Moffat (2007), combining block-wise compression with a technique called Re-Merge, the overhead achieved by repeating portions of the vocabulary in the prelude of each chunk can be reduced drastically, so the compression ratio is not damaged.

Another solution to save memory is to compute the pairs/phrases to be added in a small portion of the text and use the resulting vocabulary to compress the complete text.

Finally, the compression obtained by our new techniques can still be improved even

further on several ways. For example, by using a (s, c)-dense coding scheme (Brisaboa et al., 2007) instead of ETDC we expect improvements around 0.5 percentage points in compression ratio (and a marginal loss of speed). Also, studying new alternatives to compress the hierarchy of phrases obtained by PhETDC would be of interest.

7. Acknowledgments

This work was founded in part by Ministerio de Educación y Ciencia [TIN2009-14560-C03-02] and [TIN2010-21246-C02-01], Ministerio de Ciencia e Innovación [CDTI CEN-20091048], and Xunta de Galicia grant 2010/17 (for the Spanish group), and Fondecyt grant 1-080019 (third author). We want also to thank Ángel Yáñez Miragaya and Yolanda Varela Sobrino for their help in the implementation of PETDC and PhETDC.

References

Apostolico, A., Lonardi, S., 2000. Off-line compression by greedy textual substitution. Proceedings of the IEEE 88 (11), 1733–1744.

Bell, T. C., Cleary, J. G., Witten, I. H., 1990. Text Compression. Prentice Hall.

Bentley, J. L., Sleator, D. D., Tarjan, R. E., Wei, V. K., apr 1986. A locally adaptive data compression scheme. Communications of the ACM 29 (4).

Brisaboa, N., Fariña, A., Navarro, G., Paramá, J. R., 2006. Improving semistatic compression via pair-based coding. In: Ershov Memorial Conference; LNCS 4378. pp. 124–134.

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

Crochemore, M., 1981. An optimal algorithm for computing the repetitions in a word. Information Processing Letters 12 (5), 244–250.

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.

Huffman, D. A., 1952. A method for the construction of minimum-redundancy codes. Proc.Inst.Radio Eng. 40 (9), 1098–1101.

Katajainen, J., Raita, T., March 1 1989. An approximation algorithm for space-optimal encoding of a text. The Computer Journal 32 (3), 228–237.

Larsson, N. J., Moffat, A., 1999. Offline dictionary-based compression. In: Data Compression Conference. pp. 296–305.

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

Moura, E., Navarro, G., Ziviani, N., 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, pp. 298–306.

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

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

Rubin, F., 1976. Experiments in text file compression. Communications of the ACM 19 (11), 617–623.

Turpin, A., Smyth, W. F., 2002. An approach to phrase selection for offline data compression. In: Twenty-Fifth Australasian Computer Science Conference (ACSC2002). Vol. 4. ACS, Melbourne, Australia, pp. 267–273.

Wan, R., 2003. Browsing and searching compressed documents. Ph.D. thesis, Department of Computer Science and Software Engineering, University of Melbourne.

Wan, R., Moffat, A., 2007. Block merging for off-line compression. Journal of the American Society for Information Science and Technology 58 (1), 3–14.

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

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

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

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

Curriculums

Nieves R. Brisaboa

Nieves R. Brisaboa is the founder and director of the Database Laboratory of the University of A Coruña (http://lbd.udc.es), which counts with more than 20 researchers. As director of the laboratory, she was the main researcher of more than 30 national and international projects. She also supervised ten PhD theses. Currently, she is a full professor within the Computer Science Department of the University of A Coruña. Her research interests include digital libraries, text retrieval, compressed text retrieval, deductive databases and spatial databases.

Antonio Fariña

Antonio Fariña received his M.S. degree in Computer Science in 2000 and he earned his PhD in Computer Science in 2005 from the University of A Coruña. Today, he is an associate professor of the Computer Science Department, also he is a member of Databases Laboratory where he has been involved in different research and development projects. His research is mainly focused in text compression and indexing, searches in Metric Spaces, and Geographic Information Systems.

Gonzalo Navarro

Gonzalo Navarro is a full-professor from the University of Chile and researcher at the Millennium Institute for Cell Dynamics and Biotechnology. His areas of interest include algorithms and data structures, text searching, compression, and metric space searching. He has been (co)-chair of 6 conferences, PC member of around 45, and reviewer of around 40 international journals. He is member of the Steering Committee of LATIN and SISAP, and of the Editorial Board of journals Information Retrieval and ACM JEA. He has authored a book on string matching and around 20 book chapters, 90 papers in international journals, and 160 papers in international conferences.

José R. Paramá

José R. Paramá obtained his PhD in Computer Science in 2001 at the University of A Coruña. He is currently associate professor in the same university. His areas of interest are digital libraries, compressed text retrieval, deductive databases and spatial databases.

Tables

Strategy	Compression ratio
1	38.91 %
2	38.43 %
3	38.22 %

Table 1: Compression ratio achieved by the different strategies on the Calgary corpus.

	Size(Mb)	PH	ETDC	PETDC	PhETDC	PPMdj	p7zip	gzip	bzip2	Re-Pair
Calgary	2.0	42.16%	43.31%	41.11%	38.22%	25.36%	29.97%	36.95%	28.92%	31.20%
CR	48.72	30.41%	31.30%	27.66%	23.02%	16.88%	21.64%	33.29%	24.14%	20.15%
ZIFF	176.74	32.52%	33.41%	29.04%	23.22%	18.11%	22.99%	33.06%	25.11%	20.32%
AP	239.02	31.78%	32.60%	28.53%	23.52%	18.20%	22.78%	37.32%	27.25%	16.37%

Table 2: Compressing with PETDC and comparison in compression ratio with others.

	PH	ETDC	PETDC	PhETDC	PPMdj	p7zip	gzip	bzip2	Re-Pair
Calgary	0.14	0.15	0.46	0.61	0.80	1.61	0.26	0.42	1.85
CR	1.99	2.03	5.75	9.70	26.01	66.04	6.24	10.69	71.99
ZIFF	7.51	7.49	20.51	37.38	124.55	250.28	18.90	41.10	502.00[a]
AP	10.33	10.35	28.40	50.65	198.12	349.91	32.64	50.93	1,361.12[a]

Table 3: Compression times (seconds).

[a] These values are approximated, since our test machine was not capable of compressing these files. These measures were obtained on an Intel Xeon E5335@2.00GHz with 16GB RAM. It ran Ubuntu GNU/Linux with kernel version 2.6.24-24-server. The compiler was gcc version 4.2.4 and -O9 compiler optimizations were set.

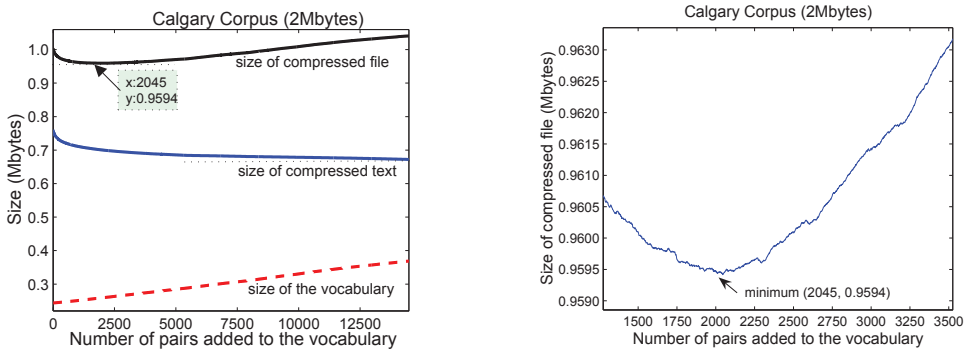
	PH	ETDC	PETDC	PhETDC	PPMdj	p7zip	gzip	bzip2	Re-Pair
Calgary	0.04	0.04	0.06	0.06	0.81	0.08	0.04	0.17	0.06
CR	0.65	0.61	0.51	0.57	28.55	1.53	0.70	3.33	1.54
ZIFF	2.40	2.32	1.92	2.45	129.66	6.11	2.92	12.28	5.96
AP	3.16	3.06	2.63	3.71	200.83	8.25	4.34	18.34	11.57

Table 4: Decompression times (seconds).

	Peak Memory usage		Size on disk
	compression	decompression	compressor executable
ETDC	38.37 MB	3.90 MB	28,163 bytes
PETDC	106.10 MB	6.97 MB	29,614 bytes
PhETDC	1.02 GB	20.09 MB	38,298 bytes
Re-Pair	1.41 GB	29.12 MB	- -

Table 5: Peak memory consumption during the compression and decompression of corpus CR; and size on disk of the executable file of the compressor for ETDC, PETDC, and PhETDC.

Figures



(a) (b)

Figure 1: Evolution of compressed file as pairs are added.

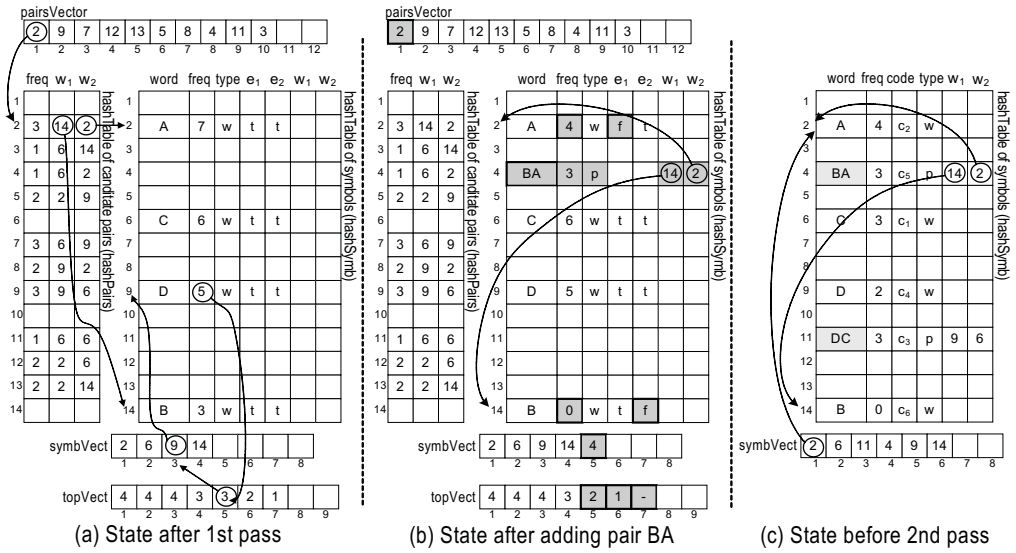


Figure 2: Structures used in PETDC for text "ADCBACDCCDACADABABACDC".

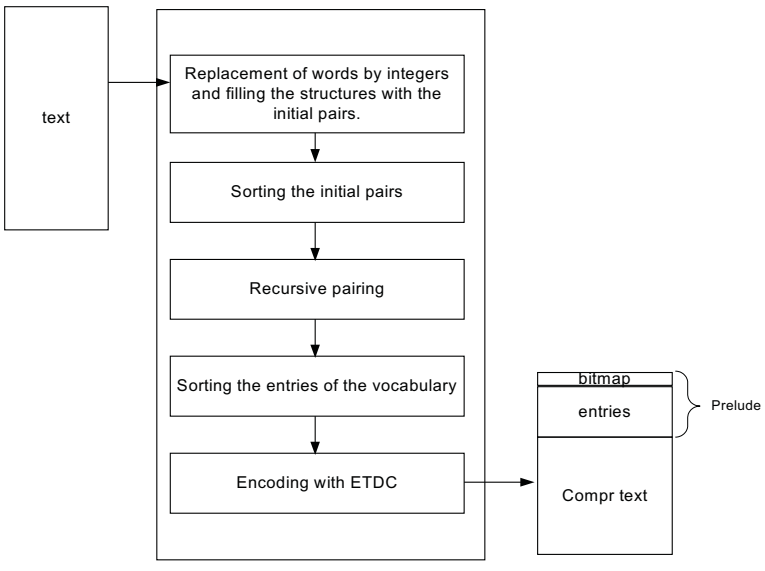


Figure 3: The PhETDC compression process.

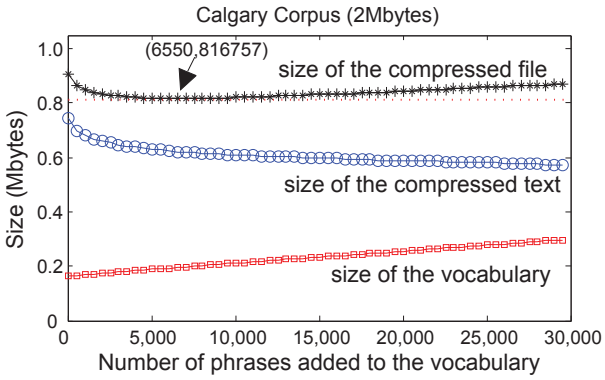


Figure 4: Evolution of the size of the compressed file as the process adds new phrases to the vocabulary.

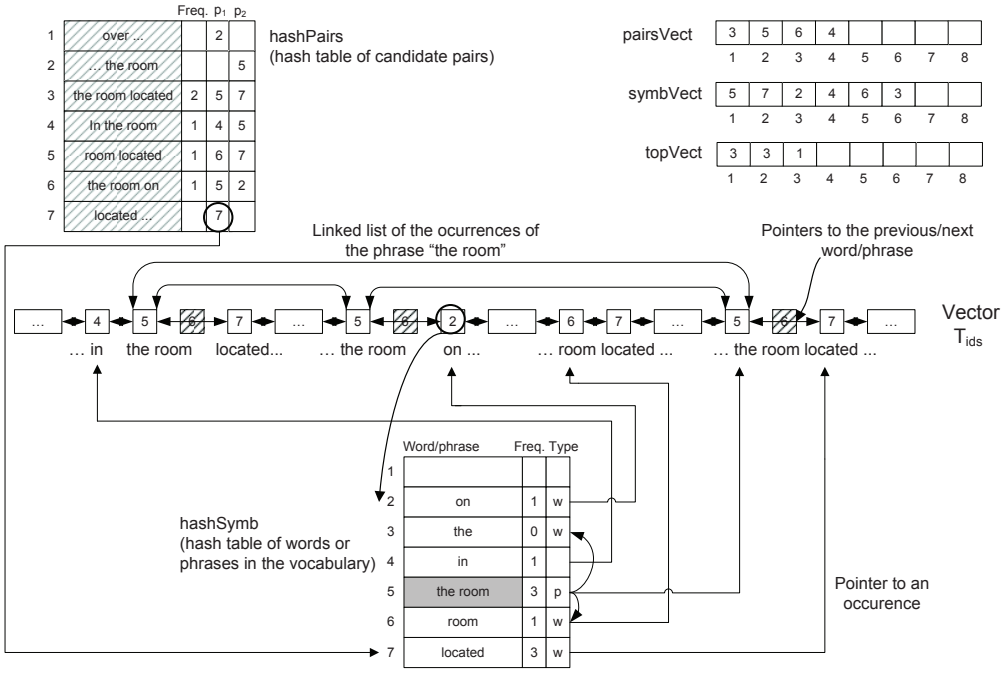
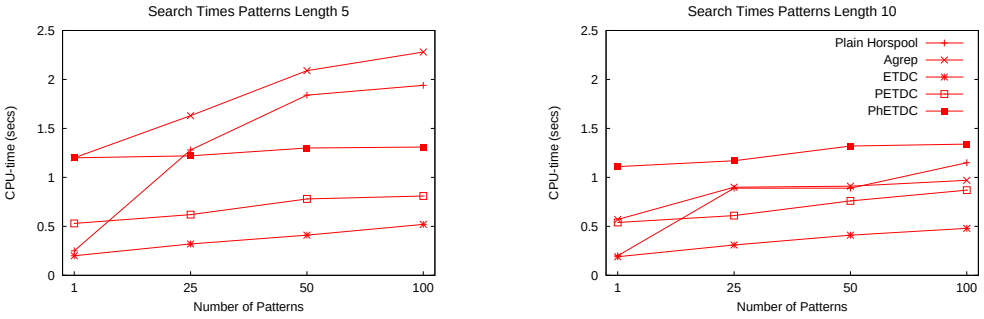
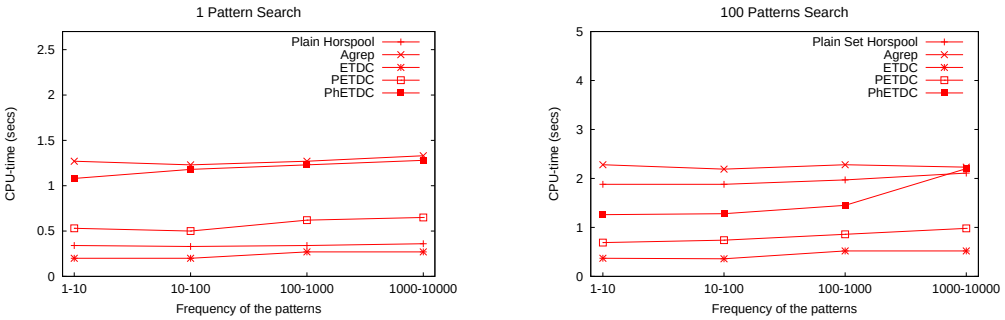


Figure 5: Structures used by the PhETDC compressor.



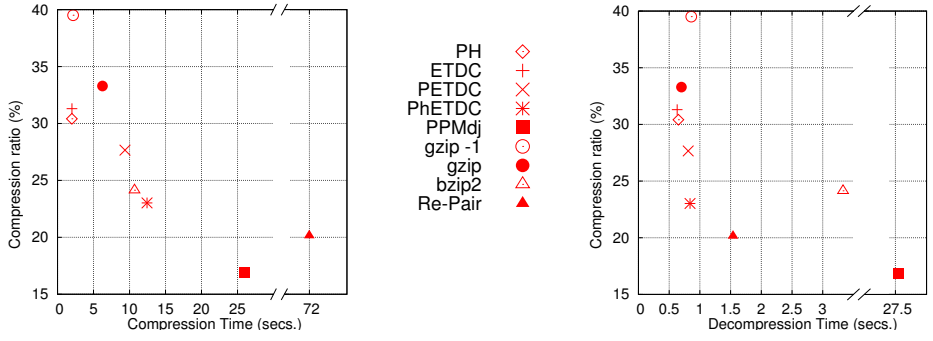
(a) (b)

Figure 6: Search times with patterns of different lengths.



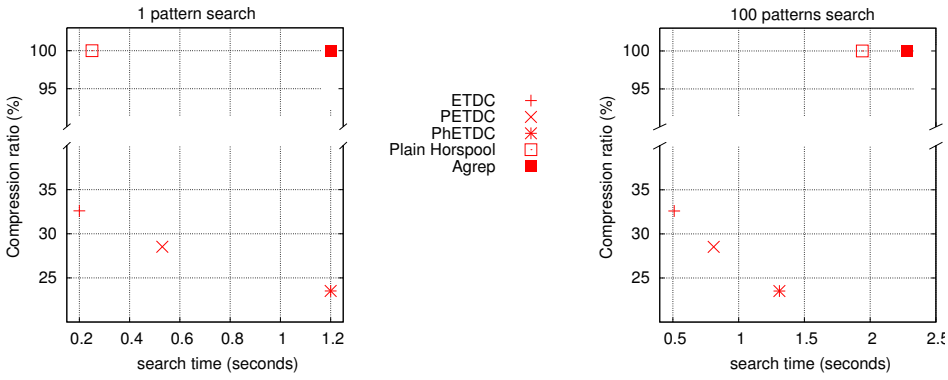
(a) (b)

Figure 7: Search time of patterns of length 5 depending on the frequency of the patterns.



(a) (b)

Figure 8: Space/time trade-offs on corpus CR, related to compression/decompression time.



(a) (b)

Figure 9: Space/Search time trade-offs on corpus AP with patterns of length 5.