Information Processing and Management 49 (2013) 392–404

Contents lists available at SciVerse ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/infoproman

DACs: Bringing direct access to variable-length codes ☆

Nieves R. Brisaboa [a], Susana Ladra [a,*], Gonzalo Navarro [b]

[a] Database Laboratory, University of Coruña, Spain
[b] Dept. of Computer Science, University of Chile, Chile

ARTICLE INFO

Article history:
Received 20 January 2012
Received in revised form 13 August 2012
Accepted 27 August 2012
Available online 8 October 2012

Keywords:
Variable length codes
Random access

ABSTRACT

We present a new variable-length encoding scheme for sequences of integers, *Directly Addressable Codes* (*DACs*), which enables direct access to any element of the encoded sequence without the need of any sampling method. Our proposal is a kind of *implicit* data structure that introduces synchronism in the encoded sequence without using asymptotically any extra space. We show some experiments demonstrating that the technique is not only simple, but also competitive in time and space with existing solutions in several applications, such as the representation of LCP arrays or high-order entropy-compressed sequences.

1. Introduction

Variable-length coding is at the heart of Data Compression (Bell, Cleary, & Witten, 1990; Moffat & Turpin, 2002; Solomon, 2007; Storer, 1988; Witten, Moffat, & Bell, 1999). It is used, for example, by statistical compression methods, which assign shorter codewords to more frequent symbols. It also arises when representing integers from an unbounded universe: Well-known codes like γ- and δ-codes are used when smaller integers are to be represented using fewer bits.

A problem that frequently arises when variable-length codes are used to encode a sequence of symbols is that it is not possible to access directly the ith encoded element, because its position in the encoded sequence depends on the sum of the lengths of the previous codewords. This is not an issue if the data is to be decoded from the beginning, as in many compression scenarios. Yet, the problem arises recurrently in the field of *compressed data structures*, where the compressed data should be randomly accessible and manipulable in compressed form. A typical scenario is that of an array of integers containing mostly small values, but still being there a few large values that prevent simply allocating a few bits for each cell.

The typical solution to provide direct access to a variable-length encoded sequence is to regularly sample it and store the position of the samples in the encoded sequence, so that decompression from the last sample is necessary. This introduces a space and time penalty to the encoding that hinders the use of variable-length coding in many cases where it would be beneficial.

In this article we present a new variable-length encoding scheme for sequences of integers that supports fast direct access to any element. This setup includes variable-length codewords coming from compression methods. We show experimentally that our technique is advantageous compared to other encoding schemes that support direct access. Our implementations are available at http://lbd.udc.es/research/DACS/.

☆ A preliminary partial version of this paper appeared in Proc. SPIRE 209, pp. 122–130.
* Corresponding author. Address: Facultad de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain. Tel.: +34 981167000; fax: +34 981167160.
E-mail addresses: brisaboa@udc.es (N.R. Brisaboa), sladra@udc.es (S. Ladra), gnavarro@dcc.uchile.cl (G. Navarro).

0306-4573/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.ipm.2012.08.003

2. Variable-length encodings

2.1. Statistical encoding

Let $X = x_1, x_2, \ldots, x_n$ be a sequence of symbols to represent. A way to compress X is to order the distinct symbol values by their frequency in X, and identify each value x_i with its position p_i in the ordering, so that smaller positions occur more frequently. Hence the problem is how to encode the p_is into variable-length bit streams $c_i = c(p_i)$ (called *codewords*), giving shorter codewords to smaller values. Huffman coding (Huffman, 1952) is the best code (i.e., achieving the minimum total length) that is univocally decodable.

2.2. Coding integers

In some applications, the x_is are directly the numbers p_i to be encoded, such that the smaller values are assumed to be more frequent. One can still use Huffman, but if the set of distinct numbers is too large, the overhead of storing the Huffman code $c(\cdot)$ may be prohibitive. In this case one can directly encode the numbers with a fixed prefix code that gives shorter codewords to smaller numbers, such as γ-codes, δ-codes, and Rice codes, to name a few (Witten et al., 1999; Solomon, 2007).

If we could assign just the minimum number of bits required to represent each number $x_i \geqslant 1$, the total length of the representation would be $N_0 = \sum_{1 \leqslant i \leqslant n}(\lfloor \log x_i \rfloor + 1)$ bits. Note that $N_0 < H + n$, where $H = \log \binom{u}{n}$ and $u = \sum_{1 \leqslant i \leqslant n} x_i$. The (at most) n extra bits of the encoding owe to the need of using an integral number of bits per code (numbers can be packed and represented as tuples to reduce this overhead). Then, γ-codes achieve $N \leqslant 2N_0$ bits by representing $\lfloor \log x_i \rfloor$ in unary and then x_i in optimal form without its most significant 1-bit. For larger numbers, δ-codes perform better by representing $\lfloor \log x_i \rfloor$ using γ-codes instead of unary codes, thus achieving $N \leqslant N_0 + 2n\log(N_0/n) + O(n)$ bits. Rice codes are parameterized by a *radix* r, so that the lowest r bits of x_i are represented verbatim, preceded by $\lfloor x_i/2_r \rfloor$ in unary.

2.3. Vbyte coding (Williams & Zobel, 1999)

This is a particularly interesting code for this article. In its general variant, the code splits the $\lfloor \log x_i \rfloor + 1$ bits needed to represent x_i into blocks of b bits and stores each block into a *chunk* of $b + 1$ bits. The highest bit is 0 in the chunk holding the most significant bits of x_i, and 1 in the rest of the chunks. For example, if $x_i = 25 = 11001_2$ and $b = 3$, we need two chunks and the representation is $\underline{0}011$ $\underline{1}001$.

Compared to an optimal encoding of $\lfloor \log x_i \rfloor + 1$ bits, Vbyte code loses one bit per b bits of x_i, plus possibly an almost empty final chunk, for a total space of $N \leqslant \lceil N_0(1 + 1/b) \rceil + nb$ bits. The best choice for the upper bound is $b = \sqrt{N_0/n}$, achieving $N \leqslant N_0 + 2n\sqrt{N_0/n}$, which is still worse than δ-encoding's performance. In exchange, Vbyte codes are very fast to decode.

2.4. Fast decodable representations

Simple9 (Anh & Moffat, 2005), Simple16 and PforDelta (Zukowski, Heman, Nes, & Boncz, 2006) are recent techniques to achieve fast decoding and little space. The general idea is to pack a number of small integers in a computer word, using the number of bits needed by the largest number. Simple9 packs the sequence into words of 32 bits. At each point of the encoding process, it regards the next numbers and computes the maximum that can be included in a word using the same number of bits for all. For example it can encode 28 1-bit numbers, 14 2-bit numbers, 9 3-bit numbers, and so on. Four bits of the 32 are reserved to encode which format was chosen for that word. Simple16 uses a similar, slightly more sophisticated, packing. In PForDelta one uses many more than 32 bits (say, 256), and treat the 10% largest numbers as exceptions that are encoded separately. In a worst-case sequence these representations may pose a very high space overhead, but in practical situations they perform very well.

3. Previous work

We now outline several solutions to the problem of giving direct access to a sequence of n concatenated variable-length codes, that is, extracting any x_i efficiently, given i. Let us call N the length in bits of the encoded sequence.

3.1. Sparse sampling

This classical solution samples the sequence and stores absolute pointers to the sampled elements, that is, to each hth element of the sequence. Access to the $(h + d)$th element, for $0 \leqslant d < h$, is done by decoding d codewords starting from the hth sample. This involves a space overhead of $\lceil n/h \rceil \lceil \log N \rceil$ bits and a time overhead of $O(h)$ to access an element, assuming we can decode each symbol in constant time.

3.2. Dense sampling (Ferragina & Venturini, 2007)

This technique represents x_i using just $\lfloor \log x_i \rfloor$ bits, and sets pointers to *every* element in the encoded sequence, which delimits each encoded value and gives constant-time access to it. By using two levels of pointers (absolute ones every $\Theta(\log N)$ values and relative ones for the rest) the extra space for the pointers is $O(n(\log\log N + \log\log M))$, where M is the maximum number stored in the sequence.

3.3. Elias–Fano-based representation (Elias, 1974; Fano, 1971)

This is a representation for a sequence y_i of n strictly increasing numbers ending at $u = y_n$. It uses $n\log(u/n) + O(n)$ bits and gives constant-time access to any y_i. The representation separates the lower $s = \lceil \log(u/n) \rceil$ bits of each element from the remaining upper bits, and stores those lower bits contiguously in an array of sn bits. The upper bits are represented in another bit array of size at most $2n$, where the bits at positions $\lfloor y_i/2_s \rfloor + i$, for each $i \leqslant n$, are set. By using more recent techniques to find the ith bit set in a bitmap in constant time and $o(n)$ extra space (Munro, 1996; Okanohara & Sadakane, 2007), any y_i can be obtained in $O(1)$ time.

This representation can be used to encode a sequence of integers $x_i \geqslant 1$, by concatenating the binary representations of numbers $x_i + 1$, and excluding their most significant 1-bits, into a bit sequence X. The starting positions of the encoded numbers in X forms a strictly increasing sequence y_i that is stored using the Elias–Fano representation described above. Thus we obtain y_i and y_{i+1} in constant time and these are the limiting positions where $x_i + 1$ (deprived of its highest 1-bit) is to be found in X. Overall, the technique achieves $O(1)$ time access and $N \leqslant N_0 + n\log(N_0/n) + O(n)$ bits of space.

3.4. Interpolative coding (Moffat & Stuiver, 2000; Teuhola, 2011)

This is a technique to encode integer sequences using variable-length codes, so that $O(\log n)$-time access is supported not only by position (as is our focus), but also by content (i.e., find the position where the sum of the x_is exceeds a threshold). The idea is to set up a virtual balanced tree over the sequence of encoded values. The encoding of a subtree is preceded by the sum of values and encoding size of the left subtree. The number of bits needed for this header is limited by the sizes of the parent headers. By letting the tree leaves handle $O(\log n)$ values, the space overhead is just $O(n\log(N)/\log(n))$ bits. The solution offers a good combination of space- and time-efficiency for both operations. However, as shown in the extensive experiments by Teuhola (2011), the codes we present in this article are orders of magnitude faster when accessing by position.

3.5. Wavelet tree (Grossi, Gupta, & Vitter, 2003)

This is a data structure that represents a sequence of symbols, and can be adapted to a variable-length representation of them (Grossi et al. showed how to adapt it to Huffman coding, but their idea can be generalized (Navarro, 2012)). Let $c(x_1)$, $c(x_2)$, ..., $c(x_n)$ be the codewords (i.e., bit streams) of the n symbols, for a prefix-free function $c(\cdot)$. The root of the wavelet tree holds a bitmap with the first bit of each codeword (n bits in total). If there are some 0-bits in this bitmap, then the left child of the root contains the second bit of those codewords starting with a 0-bit. Similarly, if there are some 1-bits in the root bitmap, then the right child of the root contains the second bit of those codewords starting with a 1-bit. This continues recursively at both children. The total number of bits in the wavelet tree is exactly N, and the wavelet tree replaces the encoded sequence.

In order to access x_i, we read the ith bit of the root bitmap B. If this bit completes a codeword, then this bit is $c(x_i)$ and we are done. If not, we continue by the left child if $B[i] = 0$, and by the right child if $B[i] = 1$. The new position at the left child is $i' = rank_0(B,i)$ and at the right child is $i' = rank_1(B,i)$, where $rank_b(B,i)$ denotes the number of occurrences of the bit b in the prefix $B[1,i]$ (we use $b = 1$ by default). Query *rank* can be computed in constant time using just $o(n)$ extra bits on top of B (Jacobson, 1989; Clark, 1996; Munro, 1996). Using such structures over all the bitmaps, the time to extract any x_i is $O(|c(x_i)|)$ and the total space is $N + o(N)$ bits, plus the pointers for the tree structure, which depend on the function $c(\cdot)$ (if this is a Huffman coding the size is equivalent to that of a Huffman table).

A variant of wavelet trees that is closely related to our approach are the Wavelet Trees on ByteCodes (Brisaboa, Faria, Ladra, & Navarro, 2008). Those also encode the sequence and provide direct access. They are, however, more complex than necessary for this application. They allow one to compute frequencies of numbers in arbitrary ranges of the sequence, and track the positions of individual values in the sequence. Our arrangement, which focuses only on direct access, is simpler and has better locality of reference.

4. Directly Addressable Codes (DACs)

4.1. Conceptual description

Given a sequence of integers $X = x_1, x_2, \ldots, x_n$, we describe our new encoding scheme that enables direct access to any element of the encoded sequence.

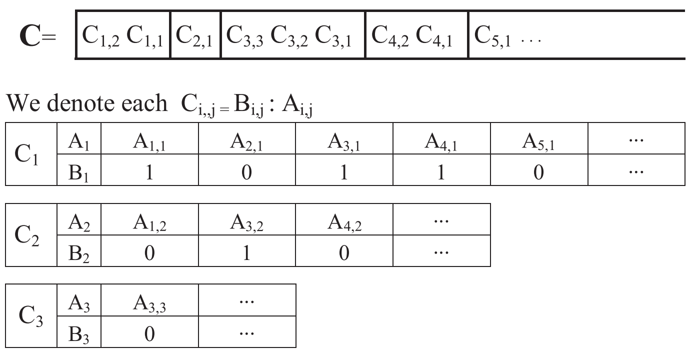

Fig. 1. Example of reorganization of the chunks of each codeword.

We make use of the generalized Vbyte coding described in Section 2. We first encode the x_is into a sequence of $(b+1)$-bit chunks. Next we separate the different chunks of each codeword. Assume x_i is assigned a codeword composed by r_i chunks $C_{i,r_i}, \ldots, C_{i,2}, C_{i,1}$. A first stream, C_1, will contain the $\mathrm{n}_1 = n$ least significant chunks (i.e., rightmost) of every codeword. A second one, C_2, will contain the n_2 second chunks of every codeword (so that there are only n_2 codewords using more than one chunk). We proceed similarly with C_3, and so on. Being M the maximum integer of the sequence, we need $L \leqslant \lceil \log(M)/b \rceil$ streams C_k.

Each stream C_k will be separated into two parts. The lowest b bits of the chunks will be stored contiguously in an array A_k (of $b \cdot n_k$ bits), whereas the highest bits will be concatenated into a bitmap B_k of n_k bits. Fig. 1 illustrates the reorganization of the chunks of a sequence of five codewords.

The bits in each B_k identify whether there is a chunk of that codeword in C_{k+1}. To find the position of the corresponding chunk in C_{k+1} we need *rank* queries on the B_k bitmap. We set up data structures on the B_k bitmaps that answer *rank* in constant time using $O\left(\frac{n_k \log\log N}{\log N}\right)$ extra bits of space, being N the length in bits of the encoded sequence.[1]

The overall structure is composed by the B_k bitmaps, their *rank* structures, the A_k arrays, and pointers to those bitmaps and arrays. These pointers need $O(\lceil \log(M)/b \rceil \lceil \log N \rceil) = O(\log M)$ bits overall, and this space is in practice negligible. In total there are $\sum_k n_k = \frac{N}{b+1}$ chunks in the encoding (note N is a multiple of $b+1$), and thus the extra space for the *rank* data structures is just $O\left(\frac{N \log\log N}{b \log N}\right) = o(N/b)$. Therefore the space is essentially that of the Vbyte representation of the sequence, plus (significantly) lower-order terms.

Extraction of the ith value of the sequence is carried out as follows. We start with $i_1 = i$ and get its first chunk $C_{i,1} = B_1[i_1]{:}A_1[i_1]$. If $B_1[i_1] = 0$ we are done with $x_i = A_1[i_1]$. Otherwise we set $i_2 = rank(B_1, i_1)$, which gives us the correct position of the second chunk of x_i in C_2, and get $C_{i,2} = B_2[i_2]{:}A_2[i_2]$. If $B_2[i_2] = 0$, we are done with $x_i = A_1[i_1] + A_2[i_2] \cdot 2^b$. Otherwise we set $i_3 = rank(B_2, i_2)$ and so on.

Extraction of a random codeword requires $\lceil N/(n(b+1)) \rceil$ accesses; the worst case is at most $\lceil \log(M)/b \rceil$ accesses. Thus, in case the numbers to represent come from a statistical variable-length coding, and the sequence is accessed at uniformly distributed positions, we have the additional benefit that shorter codewords are accessed more often and are cheaper to decode.

4.2. Implementation considerations

Our implementation uses the variant of Vbytes designed for text compression called ETDC (Brisaboa, Fariña, Navarro, & Paramá, 2007), which can make use of all the combinations of chunks and obtains slightly better space.[2] In addition, the last bitmap B_L is not stored in the final representation of the sequence of integers, since all the bits in B_L are zero.

We implement *rank* operations over the B_k bitmaps using the 5%-extra space data structure by González, Grabowski, Mäkinen, and Navarro (2005) (note that this is space over the B_k bitmaps, whose size is already $1/b$ of the total size). The times for *rank* using such extra space are of a few microseconds. If we denote ϵ the space overhead for the structure, the time for answer a *rank* operation is $O(1/\epsilon)$.

The extraction of r consecutive codewords can be performed in a more efficient way than just r independent accesses to the encoded sequence. By using one pointer at each level k of the representation, indicating the last chunk read at that level, r consecutive codewords can be extracted by computing at most $\lceil \log(M)/b \rceil$ *rank* operations to initialize those pointers and decoding sequentially the corresponding chunks at each level.

[1] This is achieved by using blocks of $\frac{1}{2}\log N$ bits in the *rank* directories (Jacobson, 1989; Clark, 1996; Munro, 1996).

[2] Note that the highest chunk of the Vbyte encoding cannot be all zeros. This wastes a combination in the highest chunk, and consequently the representation does not obtain the best possible space usage.

4.3. Minimizing the space

We have presented DACs using a fixed parameter b, which remains constant for every level of the representation. However, the value of b could be chosen differently at each level, b_k, in order to fit some goal. In particular, we can choose the b_k values with the goal of optimizing compression. This goal, however, can lead to a high number of levels L, which worsens the access time. In this section we present algorithms to choose L and b_k so that we just optimize the space without further restrictions, or we optimize space while limiting the worst-case access time L, or we optimize space while limiting the average case time.

4.3.1. Without restrictions

The optimal values can be obtained using a dynamic programming algorithm that obtains the values for L and b_k, $k \leqslant L$, that minimize the size of the representation of the given sequence.

In our dynamic programming algorithm, a subproblem t consists in encoding in the best way all the values x_i that are greater than or equal to 2^t, ignoring their t lowest bits. We start by solving trivially the case $t = m = \lfloor \log M \rfloor + 1$ and go down until $t = 0$, which represents the solution to the original problem (i.e., encoding all the numbers without ignoring bits).

Given a t, we can choose for each $t < i \leqslant m$, to encode from the tth to the $(i-1)$th bit of the numbers in a single level, and then solve the subproblem $t' = i$ using further levels. The space required per element encoded would be $i - t$ bits for sequence A_k, and $1 + \epsilon$ bits for B_k, where ϵ is the space overhead for the *rank* structure. Because the last level does not store the bitmap B_k, the cost is $(m+1) - t$ bits per element if we choose to encode from the tth to the last (mth) bits in a single level.

Algorithm 1 gives the pseudocode that obtains the optimal number of levels L and the b_k values, $k \leqslant L$. It receives m and a vector cf of cumulative frequencies of size $m+1$, that is, $cf[t]$ is the number of values x_i that are greater than or equal to 2^t. The optimal number of bits achieved for subproblem t is stored in vector $s[t]$, the corresponding number of levels in $l[t]$, and the value of b for the first level of such optimal representation in $b[t]$. The optimization costs just $O(\log^2 M)$ time (plus $O(n)$ to compute cf) and $O(\log M)$ space.

Algorithm 1. Optimize(m,cf)

```
for $t = m \cdots 0$ do
    $minSize \leftarrow +\infty$, $minPos \leftarrow m$
    for $i = t+1 \ldots m$ do
        $currentSize \leftarrow s[i] + cf[t] \cdot ((i-t) + (1+\epsilon))$
        if $minSize > currentSize$ then
            $minSize \leftarrow currentSize$, $minPos \leftarrow i$
        end
    end
    if $minSize < cf[t] \cdot ((m+1) - t)$ then
        $s[t] \leftarrow minSize$, $l[t] \leftarrow l[minPos] + 1$, $b[t] \leftarrow minPos - t$
    else
        $s[t] \leftarrow cf[t] \cdot ((m+1) - t), l[t] \leftarrow 1$, $b[t] \leftarrow (m+1) - t$
    end
end
$L \leftarrow l[0]$
$t \leftarrow 0$
for $k = 1 \ldots l[0]$ do
    $b_k \leftarrow b[t]$
    $t \leftarrow t + b[t]$
end
return $L, b_1, \ldots, b_L$
```

A byte-aligned variation of this algorithm can generate a representation of the sequence where each chunk is completely contained in a unique byte, and decompression and accesses can be implemented more efficiently in practice.

4.3.2. Limiting the number of levels to use

If we restrict the number of levels of the representation, we are limiting the worst-case access time (i.e., the access time for the maximum value of the sequence). We can obtain the optimal space restricted to using a maximum number of levels by including a new parameter in the optimization algorithm that gives the remaining number of levels available. When only one level remains, we are forced to store all the bits, from the tth to the mth, in a single level. Since the maximum number of levels is $O(\log M)$, the time complexity raises to $O(\log^3 M)$.

Table 1
Description of the LCP arrays used.

Data	Number of elements	Maximum value	Average value	Median value	Most freq. value
`dblp`	104,857,600	1,084	28.22	32	10 (2.15%)
`dna`	104,857,600	17,772	16.75	13	13 (24.59%)
`proteins`	104,857,600	35,246	195.32	6	6 (28.75%)

4.3.3. Limiting the number of rank operations

It is also possible to limit the average access time of the representation by restricting the average number of *rank* operations, or equivalently, by restricting $\sum_{k<L} n_k \leqslant C$. Once again, we are forced to create one single final level when $cf[t] > C$, else we can create one level up to the $(i-1)$th bit and solve subproblem $t' = i$ with limit $C - cf[t]$. The time becomes $O(C \log^2 M)$ time, which can be reduced by quantizing the precision of C.

5. Experimental evaluation

The Directly Addressable Codes (DACs) are practical and can be successfully used in numerous applications where direct access is required over the representation of a sequence of integers. This requirement is frequent in compressed data structures, such as suffix trees, arrays, and inverted indexes, to name just a few. We show experimentally that DACs offer a competitive alternative to other encoding schemes that support direct access.

In Section 5.1 we compare DACs with other solutions to provide direct access to sequences of integers: sparse sampling over δ-codes, γ-codes, Rice codes, Vbyte codes, Simple9, PForDelta, and Elias–Fano monotone lists. We use LCP arrays as an example of sequences of integers to encode.

Sections 5.2 and 5.3 describe scenarios where we have sequences of arbitrary symbols instead of sequences of integers. We compare the behavior of DACs in this scenario with other statistical encodings such as bit- and byte-oriented Huffman encodings, which require a sparse sampling to provide direct access over the sequence. We also compare our technique with the dense sampling of Ferragina and Venturini, explained in Section 3, and with a Huffman-shaped wavelet tree, which compactly represents a sequence of symbols from an arbitrary alphabet and supports efficient access to any element of the sequence.

For all the experiments the machine used is a AMD Phenom (tm) II X4 955 Processor (four cores) with 8 GB RAM. It ran Ubuntu GNU/Linux with kernel version 2.6.31-22-server (64 bits). We compiled with gcc version 4.4.1 and the option `-O9`.

5.1. LCP array representation

The *Longest Common Prefix (LCP)* array is a central data structure in stringology and text indexing (Manber & Myers, 1993). Consider a text $T[1,n]$ of length n, and all the suffixes of the text, that is, $T[i,n]$ with $1 \leqslant i \leqslant n$. Assume that we have all those suffixes lexicographically sorted. The LCP array stores, for each suffix, how many symbols it has in common with the previous suffix, that is, the length of the longest common prefix between each suffix and its predecessor. Most LCP values are small, but some can be very large. Hence, a variable-length encoding scheme is a good solution to represent this sequence of integers.

Our experiments were performed on 100 MB of the XML, DNA and protein texts from Pizza&Chili corpus (http://pizzachili.dcc.uchile.cl). We denote `dblp` the LCP array obtained from the XML file, which contains bibliographic information on major computer science journals and proceedings. We denote `dna` the LCP array obtained from the DNA text, which contains gene DNA sequences consisting of uppercase letters A, G, C, T, and some other few occurrences of special characters. We denote `proteins` the LCP array obtained from the protein text, which contains protein sequences where each of the 20 amino acids is coded as one uppercase letter. Some interesting information about this dataset is shown in Table 1. The first column indicates the number of elements of the LCP array. The second, third and fourth columns show, respectively, the maximum, average, and median integer values stored in the LCP array. The last column shows the most frequent integer value and its frequency.

We use several configurations for DACs. "DACs opt" stands for the alternative that uses the optimal value for b at each level of the representation without restrictions.[3] In addition, we built several configurations limiting the number of levels, denoted "DACs opt-max levels", and limiting the average cost, denoted "DACs opt-avg cost".

We compare the space and time efficiency of DACs with some integer encodings, more concretely: δ-codes, γ-codes, Rice codes using the r value that minimizes the space,[4] Simple9, PForDelta, and byte codes (Vbyte codes with $b = 7$). To support direct access over the compressed representation of the LCP array we attach a sparse sampling to the encoded sequence obtained by all these integer encoding schemes.

[3] The optimal values are b = 6, 1, 1, 1, 2 for `dblp`, b = 4, 1, 1, 1, 2, 2, 2, 2 for `dna`, and b = 3, 3, 2, 2, 2, 1, 1, 2 for `proteins`.

[4] These are $r = 5$ for `dblp`, $r = 4$ for `dna`, and $r = 7$ for `proteins`.

Table 2
Space for encoding three different LCP arrays and decompression time under different schemes. We use bold type to highlight the best values for each column.

Text	`dblp`		`dna`		`proteins`	
method	Space (bits/e)	Time (s)	Space (bits/e)	Time (s)	Space (bits/e)	Time (s)
δ-Codes	9.5421	1.04	8.3908	1.04	7.8635	1.31
γ-Codes	10.0834	1.19	7.7517	1.15	8.2899	1.40
Rice codes	6.9194	0.91	6.0493	0.89	9.5556	0.93
Simple9	7.3565	**0.17**	5.6542	**0.18**	7.6135	**0.23**
PForDelta	**6.2829**	0.18	**5.1408**	0.21	6.7323	0.33
byte codes	8.4024	0.44	8.0612	0.43	9.2683	0.51
DACs opt	7.5222	1.41	5.5434	1.35	**6.5797**	2.01

We also compare our structure with the representation of the sequence of integers using the Elias–Fano monotone lists and also using interpolative coding.[5] For the Elias–Fano representation, we use the implementation from the Sux4J project[6] (Vigna, 2008), compiling with java version 1.6.0_18.

We measure the space required by each technique in bits per element (bits/e), and decompression and access time. Decompression time measures the seconds needed to retrieve the original LCP array in plain form. Access time is measured in microseconds per access as the average time to retrieve the elements at random positions of the LCP array.

Table 2 shows the space required by δ-codes, γ-codes, Rice codes, Simple9, PForDelta and byte codes (without any sampling) to represent the three different LCP arrays, and the space occupied by "DACs opt". We also include the decompression time in seconds. We can observe that DACs obtain the best space among all the alternatives for `proteins`, while PForDelta obtains the smallest representation for the rest. Rice codes are the fastest bit-oriented alternative, while Simple9 obtains the best decompression time while achieving also very compact spaces.

The main goal of our proposal is to provide fast direct access to the encoded sequence. We tested the efficiency of DACs by accessing all the positions of each LCP array in random order. Fig. 2 shows the spaces and times achieved for `dblp` (top right), `dna` (bottom left), and `proteins` (bottom right) LCP arrays. The space for the integer encodings includes the space for the sparse sampling, where we varied the sample period to obtain the space/time trade-off[7]. We also include Elias–Fano representation and interpolative coding in this comparison. Notice that interpolative coding supports log-time access by position, but also by content, which is not efficiently supported by the rest of the encodings.

DACs obtain the most compact space among all the alternatives when using the optimal values for b on `proteins`. They are slightly superseded by PForDelta on `dna`, and clearly superseded on `dblp`, where Rice codes also achieve slightly less space. In all those cases, however, DACs are faster, by a factor of 1.4–2.0. Variants "DACs opt-max levels" and "DACs opt-avg cost" can achieve better times at the expense of worsening the compression ratio, the latter performing slightly better. The union of the DAC variants outperform all the other solutions in space and time.

5.2. High-order entropy-compressed sequences

Ferragina and Venturini (2007) gave a simple scheme (FV) to represent a sequence of symbols $S = S_1S_2 \ldots S_n$ so that it is compressed to its high-order empirical entropy and any $O(\log n)$-bit substring of S can be decoded in constant time. This is extremely useful because it permits replacing *any* sequence by its compressed variant, and any kind of access to it under the RAM model of computation retains the original time complexity. Then, the compressed representation of the sequence permits us to answer various types of queries, such as obtaining substrings or approximate queries, in efficient time without decompressing the whole compressed data.

The idea of Ferragina and Venturini is to split the sequence S of length n into *blocks* of $\frac{1}{2}\log n$ bits, and then sort the blocks by frequency. Then, each block is represented by one integer p_i: the relative position of the block among the sorted list of blocks. The next step consists in replacing each block in the sequence by the assigned integer such that a sequence of integers is obtained. Then, the sequence is stored using a dense sampling, as explained in Section 3.

For the experiments of this section, we use blocks of different sizes by regarding the texts as sequences of k-tuples, that is, we consider substrings composed of k characters as the source symbols of the text. We process the text obtaining the vocabulary of k-tuples that appear in the text, compute their frequency and sort them by frequency to obtain the p_i values. We obtain the representation of the text as the concatenation of all the codewords of the k-tuples of the text, the vocabulary of symbols and the codeword assignment if needed.[8]

[5] Thanks to J. Teuhola for providing us the code.

[6] http://sux.dsi.unimi.it/.

[7] In most encodings it is not necessary to fully decode the integers from the last sample to the one preceding the desired value; just partial information suffices to skip them. We used a distinct optimized skipping procedure for each encoding.

[8] Our DACs and Ferragina and Venturini's encoding do not require any additional information about the codeword assignment, since this assignment does not depend on the probabilities of the symbols and a dense encoding is used (the codewords are consecutively assigned). Huffman-based encodings do require the storage of the codeword assignment as they need to reconstruct the Huffman tree to properly encode and decode. However, this additional information is minimal when canonical Huffman is used, which is our case.

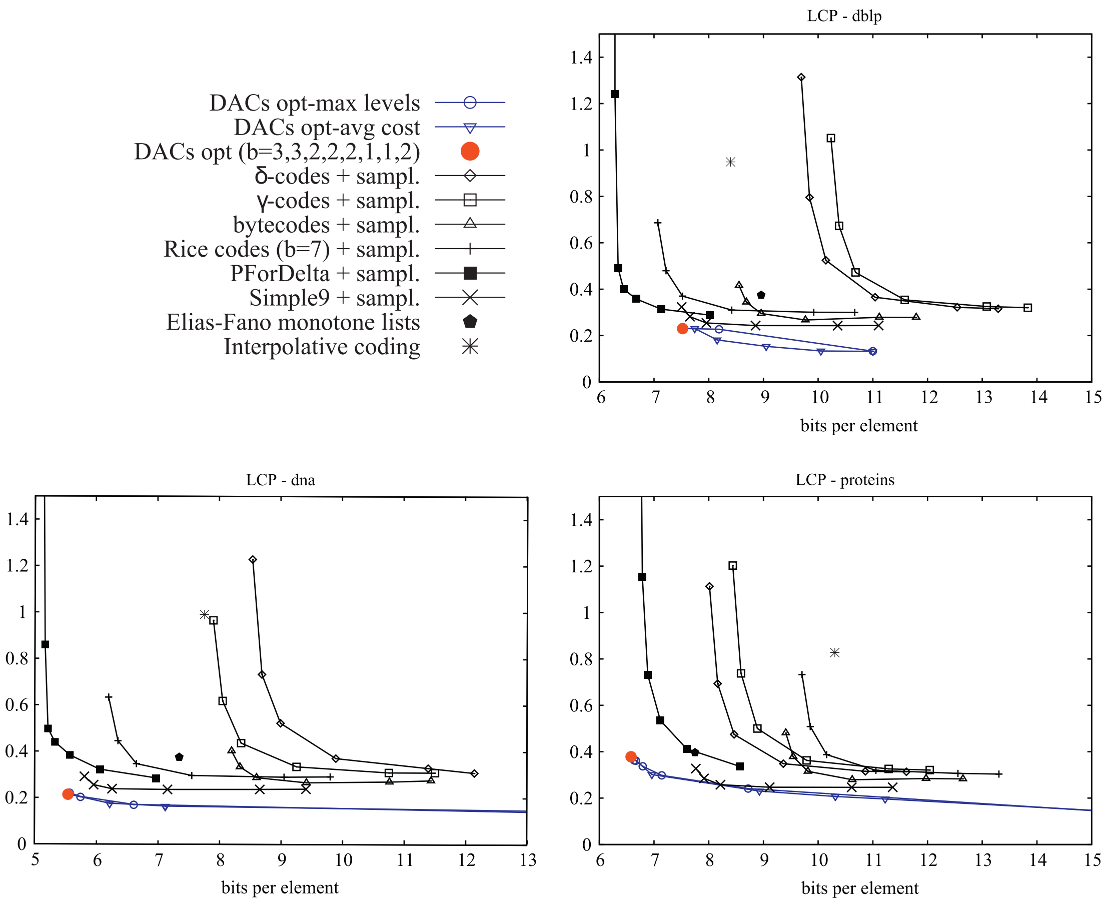

Fig. 2. Space and average access time tradeoff for different configurations of DACs and other integer encodings when accessing random positions of three LCP arrays. The y axis represents the average time per access (in µs).

Table 3
Size of the vocabulary composed of k-tuples for three different texts.

k	`xml`	`sources`	`english`
1	96	230	225
2	6676	9183	9416
3	114,643	208,235	77,617
4	585,599	1,114,490	382,398

We took the first 200 MB of three different texts from Pizza&Chili corpus. We used an XML text, denoted `xml`, containing bibliographic information on major computer science journals and proceedings.[9] We also used a text that contains source program code, denote by `sources`, formed by the concatenation of some .c, .h, .C and .java files from C and Java source code. Finally, we also used a natural language text, denoted `english`, which contains some English text files. Table 3 shows the size of the vocabulary for each text when considering tuples of length k, with k = 1–4. We compared DACs with solutions using dense and sparse sampling.

We implemented the scheme FV proposed in the paper of Ferragina and Venturini (2007), and optimized it for each scenario. Using the encoding scheme explained in Section 3, where an integer x_i is represented with $\lfloor \log x_i \rfloor$, the longest block description (corresponding to the least frequent block in the sorted vocabulary) requires a different number l of bits depending on the size of the vocabulary obtained. We use a two-level dense sampling, storing absolute pointers every c blocks and relative pointers of $\lceil \log((c-1)\cdot l) \rceil$ bits for each block inside each of those superblocks of c blocks. We adjust this setting for each text and k value to obtain the best possible space. For text `xml`, c = 20 for k = 1, 2, c = 30 for k = 3 and c = 26 for k = 4. For

[9] This XML text is the same text used to obtain the LCP array denoted `dblp` in Section 5.1.

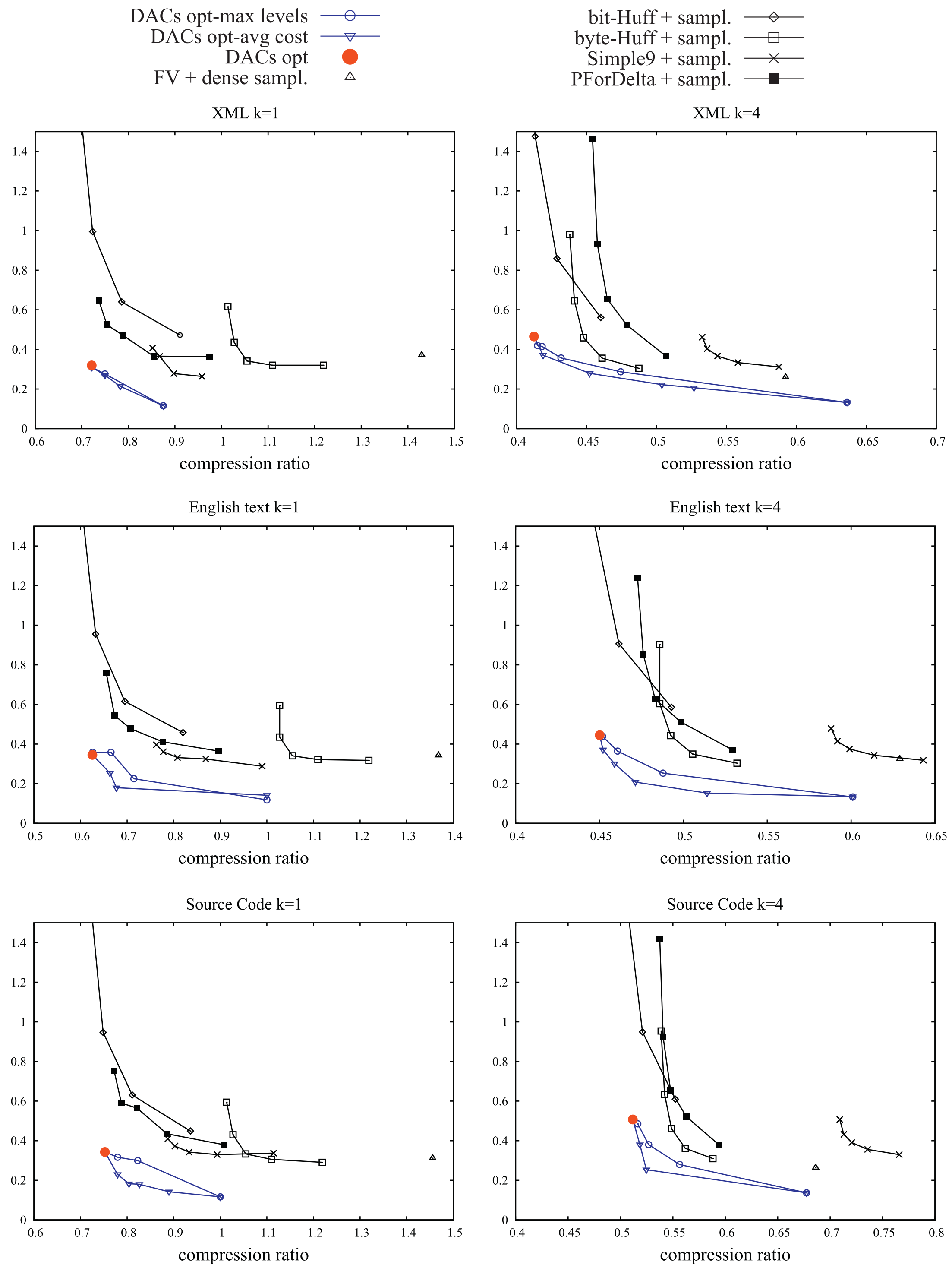

Fig. 3. Space usage and average access time for several configurations of DACs versus several encodings that represent the sequence of k-tuples for texts `xml` (top), `english` (center), and `sources` (bottom), with $k = 1$ (left) and $k = 4$ (right). The y axis represents the average time per access (in µs).

text `sources`, $c = 18$ for $k = 1,2$, $c = 30$ for $k = 3$ and $c = 24$ for $k = 4$. For text `english`, $c = 20$ for $k = 1, 2$, $c = 30$ for $k = 3$ and $c = 28$ for $k = 4$.

We also implemented the classical solution to provide direct access to any block of the sequence, by encoding the different blocks with bit-oriented and byte-oriented Huffman codes and setting absolute samples every h codewords,

$h = \{16, 32, 64, 128, 256\}$, so that partial decoding is needed to extract each value. This gives us a space–time tradeoff. We also include a Huffman-shaped wavelet tree as a solution to provide direct access to a sequence of arbitrary symbols.[10] For the comparison, we create several binary Huffman-shaped wavelet trees, varying the size for the extra structure used to compute fast binary rank operations.

We compare those solutions with several configurations of DACs. We use the b values obtained with the optimization algorithm, including the configurations where we restrict the number of levels of the representation and the average number of rank operations.

We measure the space required by each alternative in terms of compression ratio and the average access time (in microseconds per accessed k-tuple) by computing the time to access all the k-tuples of the text in random order. We illustrate in the figures the space/time tradeoff of Ferragina and Venturini's dense sampling proposal ("FV + dense sampl."), bit- and byte-oriented Huffman code plus sparse sampling ("bit-Huff + sparse sampl." and "byte-Huff + sparse sampl."), Simple9 and PForDelta plus sparse sampling ("Simple9 + sampl." and "PForDelta + sampl."), the binary Huffman-shaped wavelet tree ("huff-wt") and our DACs using the optimal b values that minimize the space ("DACs opt", "DACs opt-max levels", and "DACs opt-avg cost").

Fig. 3 shows the space/time trade-off for all the alternatives applied over the texts `xml`, `sources` and `english`, respectively, for $k = 1$ and $k = 4$. The binary Huffman-shaped wavelet tree is not shown in the figures since this method obtains high access times and its curve lies out of the plot range. We can observe that when k is increased from 1 to 4, the compression obtained is generally better, since we are compressing the text to its k-order entropy. However, if we kept increasing k, we would obtain poorer compression ratios, due to the space required to store the vocabulary. The size of the vocabulary causes that average access times are also higher for large k values for some of the solutions compared (e.g., it increases the number of levels of the representation when DACs and wavelet tree are used). Other solutions do not suffer the influence of this parameter, such as FV, where constant time is obtained due to the dense sampling.

The original FV method, implemented as such, poses much space overhead due to the dense sampling, achieving almost no compression. This, as expected, is alleviated by the bit-oriented Huffman coding with sparse sampling, but the access times increase considerably. The FV method extracts each block in constant time, while some extra decoding is always needed with the sparse sampling. Byte-oriented Huffman encoding with sparse sampling obtains better times than bit-oriented Huffman encoding, yet worsening the compression ratio. However, this byte-oriented alternative outperforms FV in space while being comparable in time. The binary Huffman-shaped wavelet tree behaves similarly to bit-oriented Huffman coding with sparse sampling for low k values, however its compression and time efficiencies degrade as the size of the vocabulary grows. Simple9 and PForDelta with sparse sampling obtain similar results to Huffman codes for $k = 1$, whereas Simple9 is not competitive for $k = 4$.

DACs improve the compression ratio when the optimal b values are computed without any restriction, adjusted according to the distribution of integers. "DACs opt" obtains a very competitive performance, and its variants dominate most of the space/time tradeoff.

As we can see, DACs can obtain good compression ratio when using the optimal b values, but sparse sampling can get slightly lower spaces. However, this comes at the price much higher access times. Hence, DACs become a very attractive solution if direct access must be provided to an encoded sequence, since it obtains very fast times and almost minimal space.

5.3. Natural language text compression

In this section we consider a sequence of integers that represents a natural language text, regarding the text as a sequence of words. The integer at position i of the sequence represents the word at position i of the text, and the integer is assigned after sorting the different words of the text by decreasing frequency, such that smaller integers are assigned to more frequent words. Various activities in a text database require to access the text at random word positions, and DACs give a direct solution to this problem. The fastest alternative is obtained when $b = 8$, that is, when bytes are used as chunks, since it avoids bitwise operations and takes advantage of the byte alignments.

We compare our fastest alternative, denoted "DACs $b = 8$", with byte-oriented Huffman encoding,[11] which is also faster than any bit-oriented encoding. As we want to directly access random words of the original text, we include a sparse sampling over the compressed sequence obtained by Plain Huffman. We denote this alternative "PH + sampl".

We used three corpora: Ziff Data 1989–1990 (ZIFF) from TREC-2, Congressional Record 1993 (CR) from TREC-4, and a large corpora (ALL), with around 1 GB, created by aggregating Ziff Data 1989–1990 (ZIFF) and AP Newswire 1988 from TREC-2, Congressional Record 1993 (CR) and Financial Times 1991–1994 from TREC-4, in addition to the small Calgary corpus.[12]

Table 4 presents the main characteristics of the corpora used. The first column indicates the name of the corpus and the second its size (in bytes). The third column indicates the number of words that compose the corpus, and finally the fourth column shows the number of different words in the text.

[10] We use the implementation available at the Compact Data Structures Library (libcds), http://libcds.recoded.cl/.

[11] The byte-oriented Huffman compressor that uses words as source symbols, instead of characters, is called *Plain Huffman* (Moura, Navarro, Ziviani, & Baeza-Yates, 2000).

[12] http://www.data-compression.info/Corpora/CalgaryCorpus/.

Table 4
Description of the corpora used.

CORPUS	Size (bytes)	Num words	Voc. size
CR	51,085,545	10,113,143	117,713
ZIFF	185,220,211	40,627,131	237,622
ALL	1,080,720,303	228,707,250	885,630

Table 5
Space and time performance for DACs and byte-oriented Huffman code (PH) when representing the sequence of words of three natural language texts.

Text	DACs $b = 8$			PH + samp			
	Ratio (%)	*t* Dec (s)	*t* Access (μs)	Ratio (%)	Words per sample	*t* Dec	*t* Access (μs)
CR	33.45	0.42	0.0544	33.53	24	0.34	0.1938
ZIFF	35.57	1.53	0.0761	35.62	26	1.26	0.2581
ALL	35.24	10.12	0.1088	35.23	32	8.57	0.2838

Table 5 shows the compression ratio (in %), decompression time (in seconds) and access time (microseconds per access) for the two alternatives over all the corpora considered. “DACs $b = 8$” uses the *rank* structure that occupies 5%-extra space over the sequence. We have adjusted the sampling parameter of the alternative “PH + sampl” to obtain the same compression ratio than “DACs $b = 8$”. The value of this parameter is shown in the table for each text: we store one sample each 24 codewords for CR corpus, one sample each 26 codewords for ZIFF corpus and one sample each 36 codewords for ALL corpus.

The decompression time includes the time, in seconds, to decompress the whole text, retrieving an exact copy of the original text. This procedure does not require the use of samples in the case of “PH + sampl”, nor does it require the use of rank operations when “DACs $b = 8$” is used, since all the levels of the representation can be sequentially processed and the synchronization between the bytes of the same codeword can be carried out using one pointer at each level, indicating the last byte read.

Decompression is faster for PH than for DACs. For PH, decompression just involves a sequential decoding of all the bytes of the encoded sequence. For DACs, it requires reading bytes at different levels of the representation, which are not contiguously located in memory, and thus have less locality of reference. In addition, the compressed sequence using PH (without taking into account the sparse sampling) is shorter than the compressed sequence using DACs. Hence, PH processes a smaller number of bytes during the decompression procedure, which also speeds up decompression.

The access time was computed as the average time to access 10,000,000 words at random positions of the text. We can observe that “DACs $b = 8$” obtains considerably better access times than “PH + sampl”, around 3–4 times faster. It is also noticeable that, for both alternatives, larger corpora obtain worse results than smaller corpora. In the case of “DACs $b = 8$” this is due to the size of the vocabulary: since there are more different words in a larger text, there are many words that obtain longer codewords, and consequently the number of levels is bigger than for smaller corpora, which causes a higher number of rank operations when extracting those codewords. In the case of “PH + sampl”, the sample period used is bigger for larger corpora, as we can see in Table 5, and this slows down the accesses to random words of the text.

Our proposal obtains better access time to individual words of the text, but it becomes slower when decompressing the whole text. We now analyze the time required by each alternative to access *t* random consecutive positions of the text, as when extracting a snippet. Fig. 4 shows the average time to retrieve *t* consecutive words for the three corpora CR, ZIFF and ALL using “DACs $b = 8$” and “PH + samp”, where the sampling used is the same as in Table 5. We observe in the figure that “DACs $b = 8$” outperforms “PH + samp” when the value of *t* is small, that is, when we access a few consecutive words of the

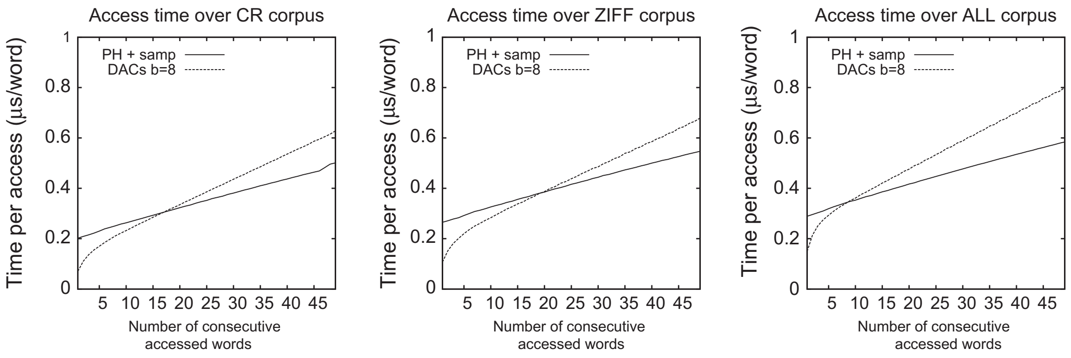

Fig. 4. Accessing consecutive words for DACs (b = 8) and PH (with sampling).

text. As we increase *t*, the benefits of PH encoding, that is, its higher locality of reference, becomes noticeable, and "PH + samp" outperforms "DACs b = 8". For instance, if we want to decompress 25 consecutive words, "PH + samp" becomes the preferred alternative. However, when accessing few consecutive words, such as five or less, "DACs b = 8" obtains better time results.

This shows that DACs may not be the best choice if we are interested in extracting snippets, which are longer than a few words. However, there is another important activity in text databases where time is more critical, and where just a few words have to be accessed: When phrases are sought, we can carry out the intersections on the inverted lists of the text collection, if they store the exact word offsets of all the occurrences of each word. However, it is usually the case that some lists are much longer than others, and better than intersecting the lists is to directly access the occurrences of the shorter list and check their surrounding words in the text. This becomes even more interesting when looking for phrases of more than two words. For this problem DACs are preferable over PH + samp.

6. Conclusions

We have introduced the *Directly Addressable Codes (DACs)*, a new storage scheme for variable-length encoded sequences (including unbounded integers and statistical encodings) that enables easy and direct access to any element of the sequence. It achieves very compact spaces, usually better than most alternative representations, and much faster direct access. This is an important achievement because the need of random access to variable-length codes is ubiquitous in many sorts of applications, particularly in compressed data structures, but also arises in everyday programming. Using bitmaps aligned to the codes in order to mark their beginnings is a folklore idea that has been used many times (Fano, 1971; Culpepper & Moffat, 2006; Fredriksson & Grabowski, 2006; Grabowski, Navarro, Przywarski, Salinger, & Mäkinen, 2006; Brisaboa et al., 2007) with various purposes, but as far as we know, our scheme to provide direct access is unique. Our method is simple to program and is space- and time-efficient, which makes it an attractive practical choice in many scenarios.

Since its original publication (Brisaboa, Ladra, & Navarro, 2009), DACs have proved to be relevant in many applications related to compact data structures:

Compressed suffix trees. A new practical compressed suffix tree (Cánovas & Navarro, 2010) includes the representation of the LCP array. DACs were used to provide fast direct access to any value of the encoded LCP array, which made the new representation faster than previous existing implementations (including some that need more space), within affordable space.

Efficient representation of grammars. DACs have also been used for representing the rules generated by Re-Pair (Larsson & Moffat, 2000) in a compressed index specialized on searching short substrings (q-grams) over highly repetitive sequences (Claude, Fariña, Martínez-Prieto, & Navarro, 2010). Specifically, DACs were used to store the lengths of the rules (most of which are short), considerably reducing the space.

Lempel–Ziv-based indexing. A text index oriented to repetitive text collections (Kreft & Navarro, 2011), based on the Lempel–Ziv 1977 parsing (Ziv & Lempel, 1977), uses various compact data structures to achieve space reductions of up to 1000-fold. DACs have been successfully used to store the skips of the tries that store the Lempel–Ziv phrases, where most skips are very short.

Direct access to grammar-compressed strings. In a practical study of dictionary representations (Brisaboa, Cánovas, Claude, Martínez-Prieto, & Navarro, 2011), Re-Pair compression of the strings was an alternative. DACs were used to regard each word, compressed into a sequence of nonterminals, as a variable-length representation of an element to which direct access was provided.

The first three applications illustrate the use of DACs as a technique to represent a sequence of numbers, most of which are expected to be small. Our own example on natural language text compression show how statistical encodings can be reduced, via sorting by frequency, to encoding a sequence of numbers as well. However, the last application listed above is different. It shows how DACs can be used, in general, to provide direct access to any encoded sequence of symbols obtained after using a variable-length encoding technique. In the case the underlying variable-length code is Vbyte (Williams & Zobel, 1999), our method can be regarded as just a reorganization of the bytes of the compressed data (plus asymptotically negligible extra space for *rank* structures), that enables direct access to it.

Acknowledgments

This work was founded in part by MCIN grant TIN2009-14560-C03-02 and CDTI CEN-20091048, and Xunta de Galicia grant 2010/17 (for the first and second authors), and Fondecyt grant 1-110066 (third author).

References

Anh, V. N., & Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval, 8*, 151–166.
Bell, T., Cleary, J., & Witten, I. (1990). *Text Compression*. New Jersey: Prentice Hall.

Brisaboa, N. R., Faria, A., Ladra, S., & Navarro, G. (2008). Reorganizing compressed text. In *Proc. of the 31th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR)* (pp. 139–146).

Brisaboa, N. R., Ladra, S., & Navarro, G. (2009). Directly addressable variable-length codes. In *Proc. 16th international symposium on string processing and information retrieval (SPIRE), LNCS* (Vol. 5721, pp. 122–130).

Brisaboa, N. R., Cánovas, R., Claude, F., Martínez-Prieto, M., & Navarro, G. (2011). Compressed string dictionaries. In *Proc. 10th international symposium on experimental algorithms (SEAs), LNCS* (Vol. 6630. pp. 136–147).

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

Cánovas, R., & Navarro, G. (2010). Practical compressed suffix trees. In *Proc. 9th international symposium on experimental algorithms (SEAs), LNCS* (Vol. 6049, pp. 94–105).

Clark, D. (1996). *Compact pat trees*. Ph.D. Thesis, Canada: University of Waterloo.

Claude, F., Fariña, A., Martínez-Prieto, M., & Navarro, G. (2010). Compressed *q*-gram indexing for highly repetitive biological sequences. In *Proc. 10th IEEE conference on bioinformatics and bioengineering (BIBE)* (pp. 86–91).

Culpepper, J. S., & Moffat, A. (2006). Phrase-based pattern matching in compressed text. In *Proc. of the 13th international symposium on string processing and information retrieval (SPIRE)* (Vol. 4209 of LNCS, pp. 337–345).

Elias, P. (1974). Efficient storage and retrieval by content and address of static files. *Journal of the ACM, 21*, 246–260.

Fano, R. (1971). *On the number of bits required to implement an associative memory*. Memo 61, Computer Structures Group, Project MAC, Massachusetts.

Ferragina, P., Venturini, R., 2007. A simple storage scheme for strings achieving entropy bounds. In *Proc. 18th annual symposium on discrete algorithms (SODAs)* (pp. 690–696).

Fredriksson, K., & Grabowski, S. (2006). A general compression algorithm that supports fast searching. *Information Processing Letters, 100*(6), 226–232.

González, R., Grabowski, S., Mäkinen, V., & Navarro, G. (2005). Practical implementation of rank and select queries. In *Poster proc. volume 4th workshop on efficient and experimental algorithms (WEAs)* (pp. 27–38).

Grabowski, S., Navarro, G., Przywarski, R., Salinger, A., & Mäkinen, V. (2006). A simple alphabet-independent FM-index. *International Journal of Foundations of Computer Science (IJFCS), 17*(6), 1365–1384.

Grossi, R., Gupta, A., & Vitter, J. (2003). High-order entropy-compressed text indexes. In *Proc. 14th annual ACM-SIAM symposium on discrete algorithms (SODAs)* (pp. 841–850).

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers (IRE), 40*(9), 1098–1101.

Jacobson, G. (1989). Space-efficient static trees and graphs. In *Proc. 30th IEEE symposium on foundations of computer science (FOCS)* (pp. 549–554).

Kreft, S., & Navarro, G. (2011). Self-indexing based on LZ77. In *Proc. 22th annual symposium on combinatorial pattern matching (CPM)* (Vol. 6661 of LNCS, pp. 41–54).

Larsson, J., & Moffat, A. (2000). Off-line dictionary-based compression. *Proceedings of the IEEE, 88*(11), 1722–1732.

Manber, U., & Myers, E. W. (1993). Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing, 22*(5), 935–948.

Moffat, A., & Stuiver, L. (2000). Binary interpolative coding for effective index compression. *Information Retrieval, 3*, 25–47.

Moffat, A., & Turpin, A. (2002). *Compression and coding algorithms*. Kluwer Academic Publishers.

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.

Munro, I. (1996). Tables. In *Proc. 16th conference on foundations of software technology and theoretical computer science (FSTTCS), LNCS* (Vol. 1180, pp. 37–42).

Navarro, G. (2012). Wavelet trees for all. In *Proc. 23rd annual symposium on combinatorial pattern matching (CPM), LNCS* (Vol. 7354, pp. 2–26).

Okanohara, D., & Sadakane, K. (2007). Practical entropy-compressed rank/select dictionary. In *Proc. 9th workshop on algorithm engineering and experiments (ALENEXs)* (pp. 60–70).

Solomon, D. (2007). *Variable-length codes for data compression*. Springer-Verlag.

Storer, J. (1988). *Data compression: Methods and theory*. Rockville, Md: Addison Wesley.

Teuhola, J. (2011). Interpolative coding of integer sequences supporting log-time random access. *Information Processing and Management, 47*, 742–761.

Vigna, S. (2008). Broadword implementation of rank/select queries. In *Proc. 5th workshop on experimental algorithms (WEAs)* (pp. 154–168).

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

Witten, I., Moffat, A., & Bell, T. (1999). *Managing gigabytes* (2nd ed.). New York: Morgan Kaufman Publishers.

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

Zukowski, M., Heman, S., Nes, N., & Boncz, P. (2006). Super-scalar ram-cpu cache compression. In *Proc. 22nd international conference on data engineering (ICDE)* (pp. 59). Washington, DC, USA: IEEE Computer Society.