Directly Addressable Variable-Length Codes

Nieves R. Brisaboa[1], Susana Ladra[1], and Gonzalo Navarro[2]

[1] Universidade da Coruña, Spain. {brisaboa|sladra}@udc.es
[2] Dept. of Computer Science, Univ. of Chile. gnavarro@dcc.uchile.cl

Abstract. We introduce a symbol reordering technique that implicitly synchronizes variable-length codes, such that it is possible to directly access the i-th codeword without need of any sampling method. The technique is practical and has many applications to the representation of ordered sets, sparse bitmaps, partial sums, and compressed data structures for suffix trees, arrays, and inverted indexes, to name just a few. We show experimentally that the technique offers a competitive alternative to other data structures that handle this problem.

1 Introduction

Variable-length coding is at the heart of Data Compression [26, 2, 27, 18, 25]. 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-codes 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 is that it is not possible to access directly the i-th 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 methods. Yet, the issue arises recurrently in the field of *compressed data structures*, where the compressed data should be accessible and manipulable in compressed form. A partial list of structures where the need to directly access variable-length codes arises includes Huffman and other similar encodings of text collections [17, 19, 3], compression of inverted lists [27, 7], compression of suffix trees and arrays (for example the Ψ function [24] and the LCP array [10]), compressed sequence representations [23, 9], partial sums [16], sparse bitmaps [23, 22, 6] and its applications to handling sets over a bounded universe supporting predecessor and successor search, and a long so on. It is indeed a common case that an array of integers contains mostly small values, but the need to handle a few large values makes programmers opt for allocating the maximum space instead of seeking for a more sophisticated solution.

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 often hinders the use of variable-length coding in many cases where it would be beneficial.

In this paper we show that, by properly reordering the target symbols of a variable-length encoding of a sequence, direct access to any codeword is easy and fast. This 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.

2 Basic Concepts

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 frequency, 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, so that shorter codewords are given to smaller values. Huffman coding [14] is the best code (i.e., achieving the minimum total length for encoding X) such that (1) assigns the same codeword to every occurrence of the same symbol, that is, if $p_i = p_j$ then $c_i = c_j$; and (2) is a prefix code, that is, no codeword is a prefix of another and thus the x_is can be decoded one after the other as soon as their codeword c_i is read. This is because the code is *instantaneous*, that is, one can tell where any codeword ends as soon as one reads its last bit.

Let N be the length in bits achieved by Huffman encoding of the sequence (disregarding the cost to store the frequency ordering of the distinct xs and the codewords assigned to the ps). Let f_j be the frequency (number of occurrences in the sequence) of the jth distinct symbol. Then $H_0 = \sum \frac{f_j}{n} \log \frac{n}{f_j}$ is called the *zero-order entropy* of X.[3] Huffman coding achieves $nH_0 \leq N < nH_0 + n$ bits.

Coding integers. In other applications, the x_is are directly the numbers p_i to be encoded, such that the smaller values are assumed to be more frequent, that is, we do not need to sort the symbols by frequency. One can still use Huffman, but if the set of distinct numbers is too large, then the overhead of storing the Huffman code may be prohibitive. In this case one can directly encode the numbers with a fixed instantaneous code that gives shorter codewords to smaller numbers. Well-known examples are γ-codes, δ-codes, and Rice codes [27, 25].

If we could assign just the minimum bits required to represent each number p_i, the total length of the representation would be $N_0 = \sum_{1 \leq i \leq n} \lfloor \log(p_i + 1) \rfloor$ bits. For example,δ-codes are instantaneous and achieve a total length $N \leq N_0 + 2n \log \frac{N_0}{n} + O(n)$ bits.

Note that if we can Huffman-encode the p_is we get $N < N_0 + n$ bits (which, remember, excludes the overhead of storing the code); yet if smaller ps are indeed more frequent than higher ones, then $N_0 \leq N$.

[3] Our logarithms are in base 2.

Vbyte coding is a particularly interesting code for this paper. In its general variant, the code splits the $\lfloor \log(p_i+1) \rfloor$ bits needed to represent p_i [4] by splitting it into blocks of b bits and storing each block into a *chunk* of $b+1$ bits. The highest bit is 0 in the chunk holding the most significant bits of p_i, and 1 in the rest of the chunks. For clarity we write the chunks from most to least significant, just like the binary representation of p_i. For example, if $p_i = 25 = 11001_2$ and $b = 3$, then we need two chunks and the representation is $\underline{0}011\ \underline{1}001$.

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

Partial sums are an extension of our problem when X is taken as a sequence of nonnegative *differences* between consecutive values of sequence $Y = y_1, y_2, \ldots y_n$, so that $y_i = sum(i) = \sum_{1 \leq j \leq i} p_j$. Hence, X is a compressed representation of Y that exploits the fact that consecutive differences are small numbers. We are then interested in obtaining efficiently $y_i = sum(i)$. Sometimes we are also interested in finding the largest $y_i \leq v$ given v, that is, $search(v) = \max\{i,\ sum(i) \leq v\}$. Let us call $S = sum(n)$ from now on.

3 Previous Work

From the previous section, we end up with a sequence of n concatenated variable-length codes. Being usually instantaneous, there is no problem in decoding them in sequence. We now outline several solutions to the problem of giving direct access to them, that is, extracting any p_i efficiently, given i.

The classical solution samples the sequence and stores absolute pointers only to the sampled elements, that is, to each h-th element of the sequence. Access to the $(h+d)$-th element, for $0 \leq d < h$, is done by decoding d codewords starting from the h-th 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. The partial sums problem is also solved by storing some sampled y_i values, which are directly accessed for *sum* or binary searched for *search*, and then summing up the p_is from the last sample.

A dense sampling is used by Ferragina and Venturini [9]. It represents p_i using just its $\lfloor \log(p_i+1) \rfloor$ bits, and sets pointers to *every* element in the encoded sequence. Thus the pointers give the ending points of the codewords. 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 $O(\frac{n \log \log N}{\log N})$, and constant-time access is possible.

[4] Recall that p_i can be the frequency-ordered position of some x_i, or directly the number x_i to be represented.

Sparse bitmaps solve the direct access and partial sums problems when the differences are strictly positive. The bitmap $B[1, S]$ has a 1 at positions y_i.

We make use of two complementary operations that can operate in constant time after building $o(S)$-bit directories on top of B [15, 5, 20]: $rank(B, i)$ is the number of 1s in $B[1, i]$, and $select(B, i)$ is the position in B of the ith 1 (similarly, $select_0(B, i)$ finds the ith 0). Then $y_i = select(B, i)$ and $search(v) = rank(B, v)$ easily solve the partial sums problem, whereas $x_i = select(B, i) - select(B, i-1)$ solves our original access problem. We can also acommodate zero-differences by setting bits $i + y_i$ in $B[1, S + n]$, so $y_i = select(B, i) - i$, $search(v) = rank(B, select_0(B, v))$, and $x_i = select(B, i) - select(B, i - 1) - 1$.

A drawback of this solution is that it needs to represent B explicitly, thus it requires $S + o(S)$ bits, which can be huge (note $S = \sum_{1 \leq i \leq n} p_i$). There has been much work on sparse bitmap representations [23, 13, 22], which ideally take space of the form $N_0 + o(N_0)$, yet constant time for both *rank* and *select* is not possible without spending a large $o(S)$ extra space term [1]. We describe next the most practical of these solutions we are aware of.

A synchronized Rice coding is used by Okanohara and Sadakane [22] in their "sada" proposal. It separates the numbers into their $l = \lfloor \log \frac{S}{n} \rfloor$ lowest bits (padding the numbers with zero bits if needed) and their highest bits. The lowest bits are stored in an array $L[1, n]$ of ln bits which is directly addressable. The numbers $\lfloor p_i/2^l \rfloor$ are represented in unary and concatenated into a bitmap $H[1, 2n]$, where they are retrieved via *select* operations: $p_i = 2^l \cdot (select(H, i) - select(H, i - 1) - 1) + L[i]$. The total space is $n \log \frac{S}{n} + O(n)$ bits and the access time is $O(1)$ since *select* can be solved in constant time over the dense bitmap H. Note that, if all the p_is are equal, then $N_0 = n \log \frac{S}{n} + O(n)$, and the encoding is space-efficient. For partial sums it achieves *sum* in constant time (as it uses *select*) and *search* in time $O(\log \frac{S}{n})$, as it solves *rank* using binary search confined to two consecutive 1s in H.

4 Our Technique: Reordered Vbytes

We make use of the generalized Vbyte coding described in Section 2. We first encode the p_is into a sequence of $(b+1)$-bit chunks. Next we separate the different chunks of each codeword. Assume p_i is assigned a codeword C_i that needs r chunks $C_{i,r}, \ldots, C_{i,2}, C_{i,1}$. A first stream, C_1, will contain the $n_1 = n$ 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. As the p_is add up to S, we need at most $\lceil \frac{\log S}{b} \rceil$ streams C_k (usually less).

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 of the chunks will be concatenated into a bitmap B_k of n_k bits. Figure 1 illustrates the reorganization of the different chunks of a sequence of five codewords. Notice that the bits in each B_k identify whether there is a chunk of that codeword in C_{k+1}.

C_k=	$C_{1,2}$ $C_{1,1}$	$C_{2,1}$	$C_{3,3}$ $C_{3,2}$ $C_{3,1}$	$C_{4,2}$ $C_{4,1}$	$C_{5,1}$ ....

Notice that each $C_{i,j} = B_{i,j} : A_{i,j}$

C_1	A_1	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$	$A_{4,1}$	$A_{5,1}$	....
	B_1	1	0	1	1	0	....

C_2	A_2	$A_{2,2}$	$A_{3,2}$	$A_{4,2}$	....
	B_2	0	1	0	....

C_3	A_3	$A_{3,3}$	....
	B_3	0	....

Fig. 1. Example of reorganization of the chunks of each codeword. We note $A_{i,j} = A_j[i]$ and $B_{i,j} = B_j[i]$.

We set up *rank* data structures on the B_k bitmaps, which answer *rank* in constant time using $O(\frac{n_k \log\log N}{\log N})$ extra bits of space, being N the length in bits of the encoded sequence[5]. Solutions to *rank* are rather practical (unlike those for *select*, despite their similar theoretical performace). In practice, excellent times are obtained using 37.5% extra space on top of B_k, and decent ones using up to 5% extra space [11, 22].

The overall structure is composed by the concatenation of the B_ks, that of the A_ks, and pointers to the beginning of the sequence of each k. These pointers need at most $\lceil \frac{\log S}{b} \rceil \lceil \log N \rceil$ bits overall, which is 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 $O(\frac{N \log\log N}{b \log N})$.

Extraction of the i-th value of the sequence is carried out as follows. We start with $i_1 = i$ and get its first chunk $b_1 = B_1[i_1] : A_1[i_1]$. If $B_1[i_1] = 0$ we are done with $p_i = A_1[i_1]$. Otherwise we set $i_2 = rank(B_1, i_1)$, which sends us to the correct position of the second chunk of p_i in B_2, and get $b_2 = B_2[i_2] : A_2[i_2]$. If $B_2[i_2] = 0$, we are done with $p_i = A_1[i_1] + A_2[i_2] \cdot 2^b$. Otherwise we set $i_3 = rank(B_2, i_2)$ and so on[6].

Extraction of a random codeword requires $\lceil \frac{N}{nb} \rceil = O(\frac{N_0}{nb})$ accesses; the worst case is at most $\lceil \frac{\log S}{b} \rceil$ accesses.

[5] This is achieved by using blocks of $\frac{1}{2} \log N$ bits in the *rank* directories [15, 5, 20].

[6] Notice that in this way we lose a value in the highest chunk, namely the one that has all the bits in zero. To avoid that, we use in our implementation the variant of Vbytes we designed for text compression called ETDC [3].

4.1 Partial Sums

The extension to partial sums is as for the classical method: We store in a vector $Y[0, n/s]$ the accumulated sum at regularly sampled positions (say every hth position). We store in $Y[j]$ the accumulated sum up to p_{hj}. The extra space required by Y is thus $\lceil n/h \rceil \lceil \log S \rceil$ bits. With those samples we can easily solve the two classic operations $sum(i)$ and $search(v)$.

We compute $sum(i)$ by accessing the last sampled $Y[j]$ before p_i, that is $j = \lfloor i/h \rfloor$ and adding up all the values between p_{hj+1} and p_i. To add those values we first sequentially add all the values between $A_1[hj+1]$ and $A_1[i]$. We compute $s_1 = hj + 1$ and $e_1 = i$ and $Acc_1 = \sum_{s_1 \le r \le e_1} A_1[r]$; then we compute $s_2 = rank(B_1, s_1 - 1) + 1$ and $e_2 = rank(B_1, e_1)$ and again $Acc_2 = \sum_{s_2 \le r \le e_2} A_2[r]$; and so on for the following levels. The final result is $Y[j] + \sum Acc_k \cdot 2^{b(k-1)}$. Notice that for a sampling step h this operation costs at most $O(\frac{h \log S}{b})$.

To perform $search(v)$ we start with a binary search for v in vector Y. Once we find the sample $Y[j]$ with the largest value not exceeding v, we start a sequential scanning and addition of the codewords until we reach v. That is, we start with $total = Y[j]$, $b_1 = hj+1$, $b_2 = rank(B_1, b_1 - 1) + 1$, $b_3 = rank(B_2, b_2 - 1) + 1$ and so on. The value of each new codeword is computed using its different chunks at levels $k = 1, 2, \ldots$, adding $A_k[b_k] \cdot 2^{b(k-1)}$ and incrementing b_k, as long as $k = 1$ or $B_{k-1}[b_{k-1} - 1] = 1$. Once computed, the value is added to *total* until we exceed the desired value v; then $search(v) = b_1 - 1$. Notice that we compute only one *rank* operation per sequence B_k, as the next chunks to read in each B_k follow the current one. The total cost for a search operation is $O(\log \frac{n}{h})$ for the binary search in the samples array plus $O(\frac{h \log S}{b})$ for the sequential addition of the codewords following the selected sample $Y[j]$.

As a practical note, the *rank*s associated to e_k in $sum(i)$ can be avoided by keeping pointers b_k and advancing them just as in our description of $search(v)$; this is convenient for not too large h, and this is the way we implement it in our experimental section.

5 Applications and Experiments

We detail now some immediate applications of our scheme, and compare it with the current solutions used in those applications. This section is not meant to be exhaustive, but rather a proof of concept, illustrative of the power and flexibility of our idea.

We implemented our technique with b values chosen manually for each level (in many cases the same b for all). We prefer powers of 2 for b, so that faster aligned accesses are possible. We implemented *rank* using the 37.5%-extra space data structure by González et al. [11] (this is space over the B_k bitmaps).

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 -O9`.

Method	Space (% of original file)	Time (nanosec per extraction)
Dense sampling (FV, $c = 20$)	94.34%	278.4
Sparse sampling ($h = 14$)	68.44%	607.6
Ours ($b = 8$)	68.46%	214.2

Table 1. Space for encoding the 2-byte blocks and individual access time under different schemes.

5.1 High-Order Compressed Sequences

Ferragina and Venturini [9] gave a simple scheme (FV) to represent a sequence of symbols $S = s_1 s_2 \ldots s_n$ 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.

The idea is to split S into *blocks* of $\frac{1}{2}\log n$ bits, and then sort the blocks by frequency. Once the sequence of their positions p_i is obtained, it is stored using a dense sampling, as explained in Section 3. We compare their dense sampling proposal with our own representation of the p_i numbers, as well as a classical variant using sparse sampling (also explained in Section 3).

We took the first 512 MB of the concatenations of collections FT91 to FT94 (Financial Times) from TREC-2 (`http://trec.nist.gov`), and chose 2-byte blocks, thus $n = 2^{29}$ and our block size is 16 bits.

We implemented scheme FV, and optimized it for this scenario. There are 5,426 different blocks, and thus the longest block description has 12 bits. We stored absolute 32-bit pointers every $c = 20$ blocks (32 bits are necessary as we address bit positions in the compressed sequence), and relative pointers of $\lceil \log((c-1) \cdot 12) \rceil = 8$ bits for each block. This was the setting giving the best space, and moreover let us manage pointers using integers and bytes, which was much faster than the general case.

We also implemented the classical alternative of Huffman-encoding the different blocks, and setting absolute samples every h codewords, so that partial decoding is needed to extract each value. This gives us a space-time tradeoff, which we set to $h = 14$ to achieve space comparable to our alternative.

We used our technique with $b = 8$, which lets us manipulate bytes and thus is faster. The space was almost the same with $b = 4$, but time was worse.

Table 1 shows the results. We measure space as a fraction of the size of the original 512 MB text (which is replaced by this structure), and time as nanoseconds per extraction, where we average over the time to extract all the blocks of the sequence in random order.

The original FV method, implemented as such, poses much space overhead (achieving almost no compression). This, as expected, is alleviated by the sparse sampling, but the access times increase considerably. Yet, our technique achieves much better space and noticeable better access times than FV. When using the same space of a sparse sampling, on the other hand, our technique is three times faster. Sparse sampling can get as low as 54% space (just the bare Huffman encoding), at the price of higher and higher access times. In general, space could be improved by considering blocks larger than 16 bits (as soon as the number of different blocks stays manageable), but as explained, our aim is to give a proof of concept.

5.2 Sparse Bitmaps

Okanohara and Sadakane [22] proposed many practical variants to represent sparse bitmaps. As explained in Section 3, this gives a solution to the partial sums problem, and to the simpler sequence access problem, when the differences are strictly positive (otherwise they need to support $select_0$ efficiently, which is not explored in their paper).

We test here their best variant, titled "sada", using their own code[7]. For our structure we show the choices $b = 4$ and $b = 8$ for all the levels. For *sum* we store absolute samples every 10 values for $b = 8$, and every 50 for $b = 4$. Their technique is definitely faster than ours for *search* (about 3 times faster for the same space), so let us focus on the simpler functionality of extracting a number from the sequence, and solving *sum*.

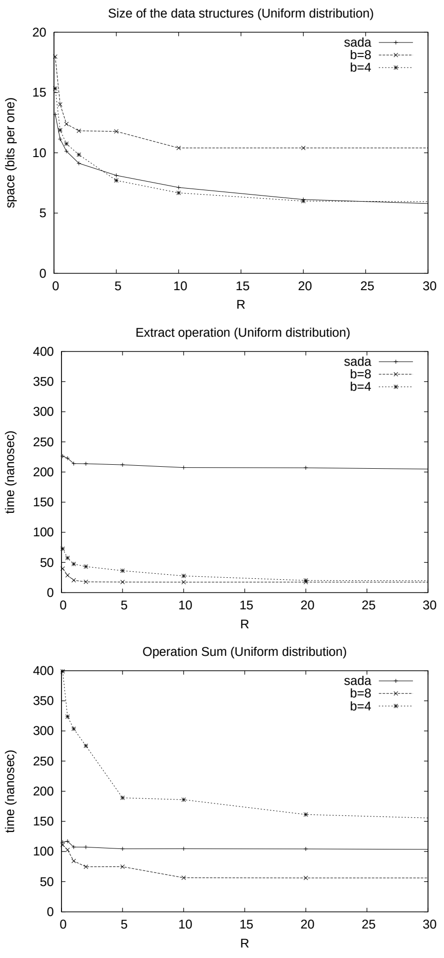
We first generate uniform bitmaps of length 10^7, with R% of 1s. Figure 2 (left) shows the space and time for both operations. We average the times of 10^7 queries. As it can be seen, our structure matches their space usage pretty well with $b = 4$, whereas we use about 50% more space with $b = 8$. For extracting, we are 5 times faster even with $b = 4$. For the more sophisticated *sum* operation, however, we are twice as fast with $b = 8$, but 1.5–4.0 times slower with $b = 4$ (where we match the space).

Their structure, however, is optimized for uniform distributions, whereas in many practical applications the distribution of the gaps is skewed. To illustrate this case, we generated bitmaps where gaps of length 1–20 are inserted with probability 90%–95%, length 300–1000 are inserted with probability 7%–2% and length 70,000–100,000 otherwise. Figure 2 (right) shows the results on this distribution. It can be seen that our structure with $b = 8$ handles much better this case. The time needed to perform sum and extract operations remains practically the same as on uniform distributions, but this time our proposal obtains better space requirements than "sada" on sparser bitmaps. We do not show the results of our proposal using $b = 4$ as it performs worse than $b = 8$ in both time and space, as large gaps are represented with a higher number of 4-bit chunks.

5.3 Compressed Suffix Arrays

Sadakane [24] proposed to represent the so-called Ψ array, useful to compress suffix arrays [12, 21], by encoding its consecutive differences along the large areas where Ψ is increasing. A γ-encoding is used to gain space, and the classical

[7] Thanks to K. Sadakane for providing us the code.



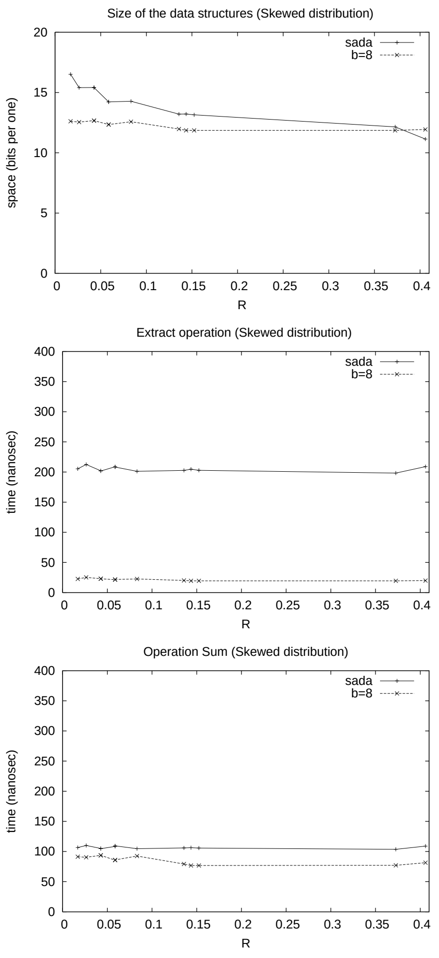


Fig. 2. Space and time versus sparse bitmap representations. On the left for uniform distributions, on the right for skewed ones. First row is space (in bits per encoded number), second is time to extract an element, and third to carry out a *sum* operation, both in nanoseconds.

Method	Space (% of original file)	Time (nanosec per Ψ computation)
Sadakane's	66.72%	645.5
Ours $b = 8$	148.06%	629.0
Ours $b = 4$	103.44%	675.6
Ours $b = 2$	85.14%	919.8
Ours $b = 0, 2, 4, 8$	73.96%	757.1
Ours* $b = 0, 2, 4, 8$	67.88%	818.7
Ours $b = 0, 4, 8$	76.85%	742.7

Table 2. Space for encoding the differential Ψ array and individual *sum* time under different schemes. The b sequences refer to the (different) consecutive b values used in the arrays C_1, C_2, etc. "Ours*" uses 5% extra space for *rank* on the bitmaps.

alternative of sampling plus decompression is used in the practical implementation. We compare now this solution to our proposal. Their implementation was obtained from *Pizza&Chili* site[8]. In these experiments, it sets one absolute sample every 128 values.

We took TREC-2 collection CR, of about 47 MB, generated its Ψ array, and measured the time to compute $\Psi^i(x)$, for $1 \leq i < n$, where x is the suffix array position pointing to the first text character. This simulates extracting the whole text by means of function Ψ without having the text at hand. Indeed, to complete the extraction we need to carry out a binary search over 256 integers for each i; we did not include this work in our measurements, as it is not related to the efficiency of computing Ψ and would just blur out the differences between the techniques.

As the differences are strictly positive, we represent in our method the differences minus 1 (so access to $\Psi[i]$ is solved via $sum(i) + i$). This time we use $b = 0$ for the first level of our structure, and other b values for the rest. This seemingly curious choice lets us spend one bit (in B_1, as A_1 is empty) to represent all the areas of Ψ where the differences are 1. This is known to be the case on large areas of Ψ for compressible texts [21], and is also a good reason for Sadakane to have chosen γ-codes. We set one absolute sample every 128 values for our *sum*. Apart from the usual *rank* version that uses 37.5% of space over the bitmaps, we tried a slower one that uses just 5% [11].

Table 2 shows the results. We measure space as a fraction of the size of the original text, and time as nanoseconds per *sum*, as this is necessary to obtain the original Ψ values from the differential version. We only show some example of fixed b, and also show how using different b values per level can achieve much better results.

This time our technique does not improve upon Sadakane's representation, which is carefully designed for this specific problem and known to be one of the best implementations [8]. Nevertheless, it is remarkable that we get rather close

[8] Mirrors `http://pizzachili.dcc.uchile.cl` and `http://pizzachili.di.unipi.it`.

(e.g., same space and 27% slower, or 15% worse in space and time) with a general and elegant technique. It is also a good opportunity to illustrate the flexibility of our technique, which lets us use different b values per level.

6 Conclusions

We have introduced a data reordering technique that, when applied to a particular class of variable-length codes, enables easy and direct access to any codeword, bypassing the heavyweight methods used in current schemes. 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. Our method is simple to program and is space- and time-efficient, which makes it an attractive practical choice in many scenarios.

We have shown experimentally that our technique competes successfully, in several immediate applications, with the methods proposed or in use for those scenarios, which gives a proof of concept of the practical value of the idea. In the future we plan to explore its suitability to other well-developed scenarios, such as the compression of inverted lists and natural language texts.

We have used the same b for every level, or manually chose it at each level to fit our applications. This could be refined and generalized to use the best b at each level, in terms of optimizing compression. The optimization problem can be easily solved by dynamic programming by noting that the subproblems are of the form "encode in the best way all the p_i values which are larger than 2^x, ignoring their 2^x lowest bits", and thus there are at most n subproblems if one sorts the p_is. As $b \leq \log S$, the optimization would cost just $O(n \log S)$ time. It is not hard also to stick to powers of 2 for b, to ensure good access times, and then the optimization costs $O(n \log \log S)$ time.

References

1. P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proc. 31st Ann. Symp. on Theory of Computing (STOC)*, pages 295–304, 1999.
2. T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, 1990.
3. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.
4. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Param. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.
5. D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Canada, 1996.
6. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th Intl. Symp. on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.
7. J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th Intl. Symp. on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 137–148, 2007.

8. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 12, 2009. 30 pages.
9. P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proc. 18th Ann. Symp. on Discrete Algorithms (SODA)*, pages 690–696, 2007.
10. J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. 19th Ann. Symp. on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 152–165, 2008.
11. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.
12. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd Symp. on Theory of Computing (STOC)*, pages 397–406, 2000.
13. A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In *Proc. 5th Intl. Workshop on Experimental Algorithms (WEA)*, pages 158–169, 2006.
14. D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1090–1101, 1952.
15. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Symp. on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.
16. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008. 38 pages.
17. A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.
18. A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.
19. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.
20. I. Munro. Tables. In *Proc. 16th Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.
21. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
22. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.
23. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Ann. Symp. on Discrete Algorithms (SODA)*, pages 233–242, 2002.
24. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
25. D. Solomon. *Variable-length codes for data compression*. Springer-Verlag, 2007.
26. J. Storer. *Data Compression: Methods and Theory*. Addison Wesley, 1988.
27. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 2nd edition, 1999.