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 [23, 21]. 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 [14, 15, 1], compression of inverted lists [23, 4], compression of suffix trees and arrays (for example the Ψ function [20] and the LCP array [7]), compressed sequence representations [19, 6], partial sums [13], sparse bitmaps [19, 18, 3] 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

* Funded in part (for the Spanish group) by MEC grant (TIN2006-15071-C03-03); and for the third author by Fondecyt Grant 1-080019, Chile.

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 (achieving constant time per symbol of the target alphabet) 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 $X = x_1 x_2 \dots x_n$ 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, giving shorter codewords to smaller values. Huffman coding [11] 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 and (2) is a prefix code.

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. One can still use Huffman, but if the set of distinct numbers is too large, the overhead of storing the Huffman code may be prohibitive. In this case one can directly encode the numbers with a fixed prefix code that gives shorter codewords to smaller numbers. Well-known examples are γ-codes and δ-codes [23, 21].

Vbyte coding [22] 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 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. Even when the best choice for b is used, the total space achieved 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 $Y = y_1, y_2, \dots y_n$, so that $y_i = sum(i) = \sum_{1 \le j \le i} p_j$. 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 $y_i \le v$ given v, that is, $search(v) = \max\{i,\ sum(i) \le v\}$. 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 prefix, 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. Let us call N the length in bits of the encoded sequence.

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 [6]. It represents p_i using just its $\lfloor \log(p_i + 1) \rfloor$ bits, and sets pointers to *every* element in the encoded sequence, giving 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.

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 [12, 2, 16]: $rank(B, i)$ 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 accommodate 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. There has been much work on sparse bitmap representations that can lighten space requirements [19, 10, 18].

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).

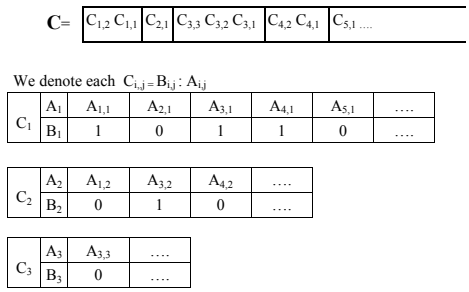

Fig. 1. Example of reorganization of the chunks of each codeword.

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. Figure 1 shows the reorganization of the different 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}.

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[3]. Solutions to *rank* are rather practical, obtaining excellent times using 37.5% extra space on top of B_k, and decent ones using up to 5% extra space [8, 18].

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[4].

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

4.1 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

[3] This is achieved by using blocks of $\frac{1}{2} \log N$ bits in the *rank* directories [12, 2, 16].

[4] To avoid the loss of a value in the highest chunk we use in our implementation the variant of Vbytes we designed for text compression called ETDC [1].

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 $Y[j]$.

5 Applications and Experiments

We detail now some 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. [8] (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`.

5.1 High-Order Compressed Sequences

Ferragina and Venturini [6] 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

Method	Space (% of original file)	Time (nanosec per extraction)
Dense sampling (FV, $c = 20$)	94.34%	298.4
Sparse sampling ($h = 14$)	68.44%	557.2
Vbyte ($b = 7$) sampling ($h = 14$)	75.90%	305.7
Ours ($b = 8$)	68.46%	216.1

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

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, 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 let us manage pointers using integers and bytes, which is faster.

We also implemented the classical alternative of Huffman-encoding the different blocks, and setting absolute samples every h codewords. This gives us a space-time tradeoff, which we set to $h = 14$ to achieve space comparable to our alternative. In addition, we implemented a variant with the same parameters but using Vbyte-encoding, with $b = 7$ (i.e., using bytes as chunks).

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, 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 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 achieve 54% space (just the bare Huffman encoding), at the price of higher access times. The Vbyte alternative is both larger and slower than ours. In fact, the Vbyte-encoding itself, without the sampling overhead, occupies 67.8% of the original sequence, very close to our representation (which will be similar to an Vbyte encoding using $b = 8$).

5.2 Compressed Suffix Arrays

Sadakane [20] proposed to represent the so-called Ψ array, useful to compress suffix arrays [9, 17], by encoding its consecutive differences along the large areas where Ψ is increasing. A γ-encoding is used to gain space, and the classical alternative of sampling plus decompression is used in the practical implementation. We compare now this solution to our proposal, using the implementation obtained from *Pizza&Chili* site[5] and setting one absolute sample every 128 values.

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

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.

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.

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 [17], 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% [8].

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 examples of fixed b, and how using different b values per level can achieve 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 [5]. Nevertheless, it is remarkable that we get rather close (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.

We have shown experimentally that our technique competes successfully, in several immediate applications. We have also compared our proposal with the best solutions for sparse bitmaps [18], but we have omitted it in Section 5 due to space limitations: Except for the *search* operation, we achieved better space and time results when the distribution of the gaps was skewed, and comparable performance otherwise (uniform distribution).

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 in just $O(n \log S)$ time.

References

1. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.
2. D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.
3. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, LNCS 5280, pages 176–187, 2008.
4. J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th SPIRE*, LNCS 4726, pages 137–148, 2007.
5. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM JEA*, 13:article 12, 2009. 30 pages.
6. P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proc. 18th SODA*, pages 690–696, 2007.
7. J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. 19th CPM*, LNCS 5029, pages 152–165, 2008.
8. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. 4th WEA*, pages 27–38, 2005.
9. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd STOC*, pages 397–406, 2000.
10. A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In *Proc. 5th WEA*, pages 158–169, 2006.
11. D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1090–1101, 1952.
12. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549–554, 1989.
13. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):1–38, 2008.
14. A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.
15. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM TOIS*, 18(2):113–139, 2000.
16. I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.
17. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
18. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th ALENEX*, 2007.
19. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
20. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
21. D. Solomon. *Variable-length codes for data compression*. Springer-Verlag, 2007.
22. H. E. Williams and J. Zobel. Compressing integers for fast file access. *COMPJ: The Computer Journal*, 42(3):193–201, 1999.
23. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 2nd edition, 1999.