Compressing Integer Sequences and Sets (2000: Moffat and Stuiver)

Alistair Moffat, The University of Melbourne,
http://www.csse.unimelb.edu.au/~alistair/

Entry editor: Paolo Ferragina

INDEX TERMS: Integer codes, difference-based coding, binary code, Elias code, Golomb code, byte-based code, Interpolative code, Packed Binary code, compressed set representations.

1 PROBLEM DEFINITION

Suppose that a message $M = \langle s_1, s_2, \ldots, s_n \rangle$ of length $n = |M|$ symbols is to be represented, where each symbol s_i is an integer in the range $1 \le s_i \le U$, for some upper limit U that may or may not be known, and may or may not be finite. Messages in this form are commonly the output of some kind of modeling step in a data compression system. The objective is to represent the message over a binary output alphabet $\{0, 1\}$ using as few as possible output bits. A special case of the problem arises when the elements of the message are strictly increasing, $s_i < s_{i+1}$. In this case the message M can be thought of as identifying a subset of $\{1, 2, \ldots, U\}$. Examples include storing sets of IP addresses or product codes, and recording the destinations of hyperlinks in the graph representation of the world wide web.

A key restriction in this problem is that it may not be assumed that $n \gg U$. That is, it must be assumed that M is too short (relative to the universe U) to warrant the calculation of an M-specific code. Indeed, in the strictly increasing case, $n \le U$ is guaranteed. A message used as an example below is $M_1 = \langle 1, 3, 1, 1, 1, 10, 8, 2, 1, 1 \rangle$. Note that any message M can be converted to another message M' over the alphabet $U' = Un$ by taking prefix sums. The transformation is reversible, with the inverse operation known as "taking gaps".

2 KEY RESULTS

A key limit on static codes is expressed by the Kraft-McMillan inequality (see [13]): if the codeword for a symbol x is of length ℓ_x, then $\sum_{x=1}^{U} 2^{-\ell_x} \le 1$ is required if the code is to be left-to-right decodeable, with no codeword a prefix of any other codeword. Another key bound is the combinatorial cost of describing a set. If an n-subset of $1 \ldots U$ is chosen at random, then a total of $\log_2 \binom{U}{n} \approx n \log_2(U/n)$ bits are required to describe that subset.

Unary and Binary codes As a first example method, consider *Unary coding*, in which the symbol x is represented as $x - 1$ bits that are `1`, followed by a single `0`-bit. For example, the first three symbols of message M_1 would be coded by "`0-110-0`", where the dashes are purely illustrative and do not form part of the coded representation. Because the Unary code for x is exactly x bits long, this code strongly favors small integers, and has a corresponding ideal symbol probability distribution (the distribution for which this particular pattern of codeword lengths yields the minimal message length) given by $\mathit{Prob}(x) = 2^{-x}$.

Unary has the useful attribute of being an infinite code. But unless the message M is dominated by small integers, Unary is a relatively expensive code. In particular, the Unary-coded representation of a message

$M = \langle s_1 \dots s_n \rangle$ requires $\sum_i s_i$ bits, and when M is a gapped representation of a subset of $1 \dots U$, can be as long as U bits in total.

The best-known code in computing is *Binary*. If $2^{k-1} < U \leq 2^k$ for some integer k, then symbols $1 \leq s_i \leq U$ can be represented in $k \geq \log_2 U$ bits each. In this case, the code is *finite*, and the ideal probability distribution is given by *Prob*$(x) = 2^{-k}$. When $U = 2^k$, this then implies that *Prob*$(x) = 2^{-\log_2 n} = 1/n$.

When U is known precisely, and is not a power of two, $2^k - U$ of the codewords can be shortened to $k - 1$ bits long, in a *Minimal Binary* code. It is conventional to assign the short codewords to symbols $1 \cdots 2^k - U$. The codewords for the remaining symbols, $(2^k - U + 1) \cdots U$, remain k bits long.

Golomb codes In 1966 Solomon Golomb provided an elegant hybrid between Unary and Binary codes (see [15]). He observed that if a random n-subset of the items $1 \cdots U$ was selected, then the gaps between consecutive members of the subset were defined by a geometric probability distribution *Prob*$(x) = p(1 - p)^{x-1}$, where $p = n/U$ is the probability that any selected item is a member of the subset.

If b is chosen such that $(1 - p)^b = 0.5$, this probability distribution suggests that the codeword for $x + b$ should be one bit longer than the codeword for x. The solution $b = \log 0.5 / \log(1-p) \approx 0.69/p \approx 0.69U/n$ specifies a parameter b that defines the *Golomb code*. To then represent integer x, calculate $1+((x-1)$ div $b)$ as a quotient, and code that part in Unary; and calculate $1 + ((x - 1) \bmod b)$ as a remainder part, and code it in Minimal Binary, against a maximum bound of b. When concatenated, the two parts form the codeword for integer x. As an example, suppose that $b = 5$ is specified. Then the five Minimal Binary codewords for the five possible binary suffix parts of the codewords are "`00`", "`01`", "`10`", "`110`", and "`111`". The number 8 is thus coded as a Unary prefix of "`10`" to indicate a quotient part of 2, followed by a Minimal Binary remainder of "`10`" representing 3, to make an overall codeword of "`10-10`".

Like Unary, the Golomb code is infinite; but by design is adjustable to different probability distributions. When $b = 2^k$ for integer k a special case of the Golomb code arises, usually called a *Rice code*.

Elias codes Peter Elias (again, see [15]) provided further hybrids between Unary and Binary codes in work published in 1975. This family of codes are defined recursively, with Unary being the simplest member.

To move from one member of the family to the next, the previous member is used to specify the number of bits in the standard binary representation of the value x being coded (that is, the value $1 + \lfloor \log_2 x \rfloor$); then, once the length has been specified, the trailing bits of x, with the top bit suppressed, are coded in Binary.

For example, the second member of the Elias family is C_γ, and can be thought of as a Unary-Binary code: Unary to indicate the prefix part, being the magnitude of x; and then Binary to indicate the value of x within the range specified by the prefix part. The first few C_γ codewords are thus "`0`", "`10-0`", "`10-1`", "`110-00`", and so on, where the dashes are again purely illustrative. In general, the C_γ codeword for a value x requires $1 + \lfloor \log_2 x \rfloor$ bits for the Unary prefix part, and a further $\lfloor \log_2 x \rfloor$ for the binary suffix part, and the ideal probability distribution is thus given by *Prob*$(x) \geq 1/(2x^2)$.

After C_γ, the next member of the Elias family is C_δ. The only difference between C_γ codewords and the corresponding C_δ codewords is that in the latter C_γ is used to store the prefix part, rather than Unary. Further members of the family of Elias codes can be generated by applying the same process recursively, but for practical purposes C_δ is the last useful member of the family, even for relatively large values of x. To see why, note that $|C_\gamma(x)| \leq |C_\delta(x)|$ whenever $x \leq 31$, meaning that C_δ is longer than the next Elias code only for values $x \geq 2^{32}$.

Fibonacci-based codes Another interesting code is derived from the Fibonacci sequence described (for this purpose) as $F_1 = 1$, $F_2 = 2$, $F_3 = 3$, $F_4 = 5$, $F_5 = 8$, and so on. The *Zeckendorf* representation of a natural number is a list of Fibonacci values that add up to that number, with the restriction that no two adjacent Fibonacci numbers may be used. For example, the number 10 is the sum of $2 + 8 = F_2 + F_5$.

The simplest *Fibonacci* code is derived directly from the ordered Zeckendorf representation of the target value, and consists of a "`0`" bit in the ith position (counting from the left) of the codeword if F_i does not appear in the sum, and a "`1`" bit in that position if it does, with indices considered in increasing order. Because it is not possible for both F_i and F_{i+1} to be part of the sum, the last two bits of this string must be "`01`". An appended "`1`" bit is thus sufficient to signal the end of each codeword. As always, the assumption

of monotonically decreasing symbol probabilities means that short codes are assigned to small values. The code for integer one is "1-1", and the next few codewords are "01-1", "001-1", "101-1", "0001-1", "1001-1", where, as before, the embedded dash is purely illustrative.

Because $F_n \approx \phi^n$ where ϕ is the golden ratio $\phi = (1 + \sqrt{5})/2 \approx 1.61803$, the codeword for x is approximately $1 + \log_\phi x \approx 1 + 1.44 \log_2 x$ bits long, and is shorter than C_γ for all values except $x = 1$. It is also as good as, or better than, C_δ over a wide range of practical values between 2 and $F_{19} = 6{,}765$. Higher-order Fibonacci codes are also possible, with increased minimum codeword lengths, and decreased coefficients on the logarithmic term. Fenwick [8] provides good coverage of Fibonacci codes.

Byte Aligned codes Performing the necessary bit-packing and bit-unpacking operations to extract unrestricted bit sequences can be costly in terms of decoding throughput rates, and a whole class of codes that operate on units of bytes rather then bits have been developed – the *Byte Aligned* codes.

The simplest Byte Aligned code is an interleaved eight-bit analog of the Elias C_γ mechanism. The top bit in each byte is reserved for a flag that indicates (when "0") that "this is the last byte of this codeword" and (when "1") that "this is not the last byte of this codeword, take another one as well". The other seven bits in each byte are used for data bits. For example, the number $1{,}234$ is coded into two bytes, "209-008", and is reconstructed via the calculation $(209 - 128 + 1) \times 128^0 + (008 + 1) \times 128^1 = 1{,}234$.

In this simplest byte aligned code, a total of $8\lceil(\log_2 x)/7\rceil$ bits are used, which makes it more effective asymptotically than the $1 + 2\lfloor \log_2 x \rfloor$ bits required by the Elias C_γ code. However, the minimum codeword length of eight bits means that Byte Aligned codes are expensive on messages dominated by small values.

Byte Aligned codes are fast to decode. They also provide another useful feature – the facility to quickly "seek" forwards in the compressed stream over a given number of codewords. A third key advantage of byte codes is that if the compressed message is to be searched, the search pattern can be rendered into a sequence of bytes using the same code, and then any byte-based pattern matching utility be invoked [7]. The zero top bit in all final bytes means that false matches are identified with a single additional test.

An improvement to the simple Byte Aligned coding mechanism arises from the observation that there is nothing special about the value 128 as the separating value between the *stopper* and *continuer* bytes, and that different values lead to different tradeoffs in overall codeword lengths [3]. In these (S, C)-Byte Aligned codes, values of S and C such that $S + C = 256$ are chosen, and each codeword consists of a sequence of zero or more continuer bytes with values greater than or equal to S, and ends with a final stopper byte with a value less than S. Other variants include methods that use bytes as the coding units to form Huffman codes, either using eight-bit coding symbols or tagged seven-bit units [7]; and methods that partially permute the alphabet, but avoid the need for a complete mapping [6]. Culpepper and Moffat [6] also describe a byte aligned coding method that creates a set of byte-based codewords with the property that the first byte uniquely identifies the length of the codeword. Similarly, *Nibble* codes can be designed as a 4-bit analog of the Byte Aligned approach, where one bit is reserved for a stopper-continuer flag, and three bits are used for data.

Other static codes There have been a wide range of other variants described in the literature. Several of these adjust the code by altering the boundaries of the set of *buckets* that define the code, and coding a value x as a Unary bucket identifier, followed by a Minimal Binary offset within the specified bucket (see [15]).

For example, the Elias C_γ code can be regarded as being a Unary-Binary combination relative to a vector of bucket sizes $\langle 2^0, 2^1, 2^2, 2^3, 2^4, \ldots \rangle$. Teuhola (see [15]) proposed a hybrid in which a parameter k is chosen, and the vector of bucket sizes is given by $\langle 2^k, 2^{k+1}, 2^{k+2}, 2^{k+3}, \ldots \rangle$. One way of setting the parameter k is to take it to be the length in bits of the median sequence value, so that the first bit of each codeword approximately halves the range of observed symbol values. Another variant method is described by Boldi and Vigna [2], who use a vector $\langle 2^k - 1,\ (2^k - 1)2^k,\ (2^k - 1)2^{2k},\ (2^k - 1)2^{3k}, \ldots \rangle$ to obtain a family of codes that are analytically and empirically well-suited to power-law probability distributions, especially those associated with web-graph compression. In this method k is typically in the range 2 to 4, and a Minimal Binary code is used for the suffix part.

Fenwick [8] provides detailed coverage of a wide range of static coding methods. Chen *et al.* [4] have also recently considered the problem of coding messages over sparse alphabets.

Index i	Value s_i	*lo*	*hi*	*lo′*	*hi′*	$s_i - lo', hi' - lo'$	Binary	MinBin
5	7	1	29	5	24	2,19	00010	0010
2	4	1	6	2	4	2, 2	10	11
1	1	1	3	1	3	0, 2	00	0
3	5	5	6	5	5	0, 0	--	--
4	6	6	6	6	6	0, 0	--	--
8	27	8	29	10	27	17,17	01111	11111
6	17	8	26	8	25	9,17	01001	1001
7	25	18	26	18	26	7, 8	0111	1110
9	28	28	29	28	28	0, 0	--	--
10	29	29	29	29	29	0, 0	--	--

Table 1: Example encodings of message $M_2 = \langle 1, 4, 5, 6, 7, 17, 25, 27, 28, 29\rangle$ using the Interpolative code. When a Minimal Binary code is used, a total of 20 bits are required. When $lo' = hi'$, no bits are output.

A context sensitive code The static codes described in the previous sections use the same set of codeword assignments throughout the encoding of the message. Better compression can be achieved in situations in which the symbol probability distribution is locally homogeneous, but not globally homogeneous.

Moffat and Stuiver [12] provided an off-line method that processes the message holisticly, in this case not because a parameter is computed (as is the case for the Binary code), but because the symbols are coded in a non-sequential manner. Their *Interpolative* code is a recursive coding method that is capable of achieving very compact representations, especially when the gaps are not independent of each other.

To explain the method, consider the subset form of the example message, as shown by sequence M_2 in Table 1. Suppose that the decoder is aware that the largest value in the subset does not exceed 29. Then every item in M is greater than or equal to $lo = 1$ and less than or equal to $hi = 29$, and the 29 different possibilities could be coded using Binary in fewer than $\lceil \log_2(29 - 1 + 1)\rceil = 5$ bits each. In particular, the mid-value in M_2, in this example the value $s_5 = 7$ (it doesn't matter which mid-value is chosen), can certainly be transmitted to the decoder using five bits. Then, once the middle number is pinned down, all of the remaining values can be coded within more precise ranges, and might require fewer than five bits each.

Now consider in more detail the range of values that the mid-value can span. Since there are $n = 10$ numbers in the list overall, there are four distinct values that precede s_5, and another five that follow it. From this argument a more restricted range for s_5 can be inferred: $lo' = lo + 4$ and $hi' = hi - 5$, meaning that the fifth value of M_2 (the number 7) can be Minimal Binary coded as a value within the range $[5, 24]$ using just 4 bits. The first row of Table 1 shows this process.

Now there are two recursive subproblems – transmitting the left part, $\langle 1, 4, 5, 6\rangle$, against the knowledge that every value is greater than $lo = 1$ and $hi = 7-1 = 6$; and transmitting the right part, $\langle 17, 25, 27, 28, 29\rangle$, against the knowledge that every value is greater than $lo = 7 + 1 = 8$ and less than or equal to $hi = 29$. These two sublists are processed recursively in the order shown in the remainder of Table 1, again with tighter ranges $[lo', hi']$ calculated and Minimal Binary codes emitted

One key aspect of the Interpolative code is that the situation can arise in which codewords that are zero bits long are called for, indicated when $lo' = hi'$. No bits need to be emitted in this case, since only one value is within the indicated range and the decoder can infer it. Four of the symbols in M_2 benefit from this possibility. This feature means that the Interpolative code is particularly effective when the subset contains clusters of consecutive items, or localized subset regions where there is a high density. In the limit, if the subset contains every element in the universal set, no bits at all are required once U is known. More generally, it is possible for dense sets to be represented in fewer than one bit per symbol.

Table 1 presents the Interpolative code using (in the final column) Minimal Binary for each value within its bounded range. A refinement is to use a Centered Minimal Binary code so that the short codewords are assigned in the middle of the range rather than at the beginning, recognizing that the mid value in a set is more likely to be near the middle of the range spanned by those items than it is to the ends of the range.

Adding this enhancement requires a trivial restructure of Minimal Binary coding, and tends to be beneficial in practice. But improvement is not guaranteed, and, as it turns out, on sequence M_2 the use of a Centered Minimal Binary code adds one bit to the length of the compressed representation compared to the Minimal Binary code shown in Table 1.

Cheng *et al.* [5] describe in detail techniques for fast decoding of Interpolative codes.

Hybrid methods It was noted above that the message must be assumed to be short relative to the total possible universe of symbols, and that $n \ll U$. Fraenkel and Klein [9] observed that the sequence of symbol *magnitudes* (that is, the sequence of values $\lceil \log_2 s_i \rceil$) in the message must be over a much more compact and dense range than the message itself, and it can be effective to use a principled code for the prefix parts that indicate the magnitude, in conjunction with straightforward Binary codes for the suffix parts. That is, rather than using Unary for the prefix part, a Huffman (minimum-redundancy) code can be used.

In 1996 Peter Fenwick (see [13]) described a similar mechanism using Arithmetic coding, and as well incorporated an additional benefit. His *Structured Arithmetic* coder makes use of adaptive probability estimation and two-part codes, being a magnitude and a suffix part, with both calculated adaptively. The magnitude parts have a small range, and that code is allowed to adapt its inferred probability distribution quickly, to account for volatile local probability changes. The resultant two-stage coding process has the unique benefit of "smearing" probability changes across ranges of values, rather than confining them to the actual values recently processed.

Other coding methods Other recent context sensitive codes include the *Binary Adaptive Sequential* code of Moffat and Anh [11]; and the *Packed Binary* codes of Anh and Moffat [1]. More generally, Witten *et al.* [15] and Moffat and Turpin [13] provide details of the Huffman and Arithmetic coding techniques that are likely to yield better compression when the length of the message M is large relative to the size of the source alphabet U.

3 APPLICATIONS

A key application of compressed set representation techniques is to the storage of inverted indexes in large full-text retrieval systems of the kind operated by web search companies [15].

4 OPEN PROBLEMS

There has been recent work on compressed set representations that support operations such as *rank* and *select*, without requiring that the set be decompressed (see, for example, Gupta *et al.* [10, 14]). Improvements to these methods, and balancing the requirements of effective compression versus efficient data access, are active areas of research.

5 EXPERIMENTAL RESULTS

Comparisons based on typical data sets of a realistic size, reporting both compression effectiveness and decoding efficiency are the norm in this area of work. Witten *et al.* [15] give details of actual compression performance, as do the majority of published papers.

6 DATA SETS

None is reported.

7 URL to CODE

The page at `http://www.csse.unimelb.edu.au/~alistair/codes/` provides a simple text-based "compression" system that allows exploration of the various codes described here.

8 CROSS REFERENCES

Arithmetic coding, Compressed full-text indexing, Rank and Select over Binary Strings.

9 RECOMMENDED READING

[1] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, June 2006.

[2] P. Boldi and S. Vigna. Codes for the world-wide web. *Internet Mathematics*, 2(4):405–427, 2005.

[3] N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In M. A. Nascimento, editor, *Proc. Symp. String Processing and Information Retrieval*, pages 122–136, Manaus, Brazil, October 2003. LNCS Volume 2857.

[4] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Optimal alphabet partitioning for semi-adaptive coding of sources of unknown sparse distributions. In J. A. Storer and M. Cohn, editors, *Proc. 2003 IEEE Data Compression Conference*, pages 372–381. IEEE Computer Society Press, Los Alamitos, California, March 2003.

[5] C.-S. Cheng, J. J.-J. Shann, and C.-P. Chung. Unique-order interpolative coding for fast querying and space-efficient indexing in information retrieval systems. *Information Processing & Management*, 42(2):407–428, March 2006.

[6] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In M. P. Consens and G. Navarro, editors, *Proc. Symp. String Processing and Information Retrieval*, pages 1–12, Buenos Aires, November 2005. LNCS Volume 3772.

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

[8] P. Fenwick. Universal codes. In K. Sayood, editor, *Lossless Compression Handbook*, pages 55–78, Boston, 2003. Academic Press.

[9] A. S. Fraenkel and S. T. Klein. Novel compression of sparse bit-strings—Preliminary report. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, Volume 12*, NATO ASI Series F, pages 169–183, Berlin, 1985. Springer-Verlag.

[10] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. In J. A. Storer and M. Cohn, editors, *Proc. 16th IEEE Data Compression Conference*, pages 213–222, Snowbird, Utah, March 2006. IEEE Computer Society, Los Alamitos, CA.

[11] A. Moffat and V. N. Anh. Binary codes for locally homogeneous sequences. *Information Processing Letters*, 99(5):75–80, September 2006. Source code available from `www.cs.mu.oz.au/~alistair/rbuc/`.

[12] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.

[13] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, Boston, MA, 2002.

[14] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, San Francisco, CA, January 2002. SIAM, Philadelphia, PA.

[15] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, second edition, 1999.