Large-Alphabet Semi-Static Entropy Coding Via Asymmetric Numeral Systems

ALISTAIR MOFFAT, The University of Melbourne
MATTHIAS PETRI, The University of Melbourne

An entropy coder takes as input a sequence of symbol identifiers over some specified alphabet and represents that sequence as a bitstring using as few bits as possible, typically assuming that the elements of the sequence are independent of each other. Previous entropy coding methods include the well-known Huffman and arithmetic approaches. Here we examine the newer asymmetric numeral systems (ANS) technique for entropy coding, and develop mechanisms that allow it to be efficiently used when the size of the source alphabet is large – thousands or millions of symbols. In particular, we examine different ways in which probability distributions over large alphabets can be approximated, and in doing so infer techniques that allow the ANS mechanism to be extended to support large-alphabet entropy coding. As well as providing a full description of ANS, we also present detailed experiments using several different types of input, including data streams arising as typical output from the modeling stages of text compression software; and compare the proposed ANS variants with Huffman and arithmetic coding baselines, measuring both compression effectiveness, and also encoding and decoding throughput. We demonstrate that in applications in which semi-static compression is appropriate, ANS-based coders can provide an excellent balance between compression effectiveness and speed, even when the alphabet is large.

CCS Concepts: •**Theory of computation** →**Data compression;** •**Information systems** →*Data compression;* Search index compression; •**Mathematics of computing** →Coding theory;

Additional Key Words and Phrases: Asymmetric numeral systems; compression; Burrows-Wheeler transform; entropy coder; Huffman code; arithmetic code

ACM Reference format:
Alistair Moffat and Matthias Petri. 2020. Large-Alphabet Semi-Static Entropy Coding Via Asymmetric Numeral Systems. *ACM Transactions on Information Systems* 1, 1, Article 1 (January 2020), 33 pages.
DOI: 10.1145/3397175

1 INTRODUCTION

An entropy coder takes as input an m-sequence of symbol identifiers over an n-symbol source alphabet, and represents that m-sequence as a bitstring using as few bits as possible, typically assuming that the elements of the sequence are independent of each other [2, 3, 29]. One obvious option is to use binary codes, with each element in the universe of n different symbols coded in $\lceil \log_2 n \rceil$ bits. While simple, this approach is effective only when all of the symbols have approximately equal likelihood of occurring.

Huffman's famous technique [28, 44] provides for the construction of variable-length codes, where the length of the binary codeword assigned to each of the n symbols is approximately inversely logarithmicly proportional to the probability of that symbol occurring, and hence provides compact

message representations when the symbol distributions are non-uniform. Three decades later, the development of multi-symbol arithmetic coding [33, 46] allowed an even closer fit between the symbol probability distribution and the generated message length; arithmetic coding also proved to be a more versatile tool than Huffman coding, in that it allowed multi-context adaptive compression models to prove their worth, facilitating the development of a whole genre of high-effectiveness compression techniques [9, 11, 26].

More recently, a third technique for entropy coding has been developed – the *asymmetric numeral systems* (ANS) approach of Jarosław (Jarek) Duda [15, 16]. Like arithmetic coding, in pure form ANS converts a sequence of input symbols into a single number that encapsulates a description of the sequence; unlike arithmetic coding, ANS generates an integer value in which each additional message symbol increases the magnitude of the value, rather than (in arithmetic coding) narrowing the range spanned by an increasing-length fractional value.

Contributions. Given that context, our goals with this work are:

- To provide a complete and accessible description of both the *range ANS* and *table ANS* approaches [15, 16], including the renormalization processes that allow ANS to be implemented without requiring arbitrary-precision operations;
- To present additional techniques that allow ANS to be used in large-alphabet applications, where n, the number of distinct symbols in the message, might range into the millions, and where the message length m might also be millions; and
- To measure the relative effectiveness and efficiency of ANS in such applications, compared to previous large-alphabet implementations of Huffman and arithmetic coding.

The next section provides definitions that specify the problem that is addressed by all three of these entropy coding methods, and provides an overview of the two classical methods: Huffman and arithmetic coding. Section 3 then provides a detailed description of ANS coding, the newest of the three entropy coding methods, addressing the first of the goals listed above. Issues that arise when ANS is applied to large-alphabet coding situations (the second goal) are examined in Section 4; and Section 5 compares ANS with Huffman and arithmetic coding in terms of effectiveness and efficiency, on both natural and synthetic data streams (the third objective). Section 6 then concludes our presentation.

2 BACKGROUND

We first provide an overview of data compression techniques, and then briefly introduce the two established methods for carrying out entropy coding.

2.1 Compression Models

Any given compression system is a composition of several phases, each of which might be exchanged for alternative mechanisms. In particular, text and data compression algorithms are typically regarded as consisting of three inter-related components; see, for example, Moffat and Turpin [36]. The first *modeling* component seeks to infer the underpinning structure of the input sequence. For example, if the symbols in the sequence are independent of each other and were generated from a single-state memoryless source, then a zero-order model is likely to be sufficient to capture all of the redundancy. On the other hand, typical natural language text shows a very broad range of inter-symbol dependencies and correlations, and high-order (or multi-state) models are a better fit. Each state (or *context*) within the model then gives rise to its own distinctive set of probabilities, presuming encoder and decoder have available to them sufficient information that they can always

agree as to which state they are in. That is, each input symbol might be coded in any one of the contexts, depending on decisions made in synchrony in the encoder and decoder.

The second component is that of *probability estimation*. Given a particular context, and the sub-sequence of symbols to be coded relative to it, each possible symbol needs to be given an estimated probability of occurrence. The simplest way in which this can be done is to count the fraction of previous uses of that context that were associated with each symbol, and develop an empirical estimate. But allowance also needs to be made for new hitherto-unseen symbols. If the model is a good fit for the data at hand, the sub-sequence of symbols associated with each context will behave as if it was generated by a single-state memoryless source, and the probability estimates will converge towards an eventual fixed point.

The third component of every compression system is *entropy coding*, the focus of this work. Given probability estimates, and a sequence of symbols assumed to be emitted by a one-state memoryless source, the task of the entropy coder is to represent that sequence as a bitstream (or output symbols using some larger non-binary channel alphabet) that is as short as possible. To allow the coding component be discussed in isolation, in the remainder of this section and throughout Sections 3 and 4 we suppose that only one context is needed. Similarly, the test data discussed in Section 5 is drawn from one context of more complex compression models.

Static, Semi-Static, and Adaptive Probability Estimation. Probability estimation can be done via *static* probabilities that are independent of the sequence being coded; can be done in a *semi-static* manner based on a first pass through the whole sequence; or can be done using *adaptive* probability estimates, computed and refined on-the-fly as each element of the input sequence is handled. Again, see Moffat and Turpin [36] for further explanation. In this work our primary focus is on semi-static estimation. In these approaches the input sequence is processed once in a "count the symbols" mode; next those counts are used as the basis of a computation to construct an entropy code; and then in a second pass the sequence is re-processed to generate bits, using a code that depends on the particular sequence, but is fixed for the whole of the sequence. Note that in this approach the symbol frequencies – or some surrogate for them – must be communicated to the decoder in a message *prelude*, so that it can construct the same fixed code.

Natural Probability Distributions. A range of compression situations fit this mode of operation. For example, in a word-based scheme, natural language is parsed into words, and the stream of words is represented against a probability distribution [13, 48]. In this case the input alphabet is the set of indices into a vocabulary data structure. Another natural alphabet arises as part of Burrows-Wheeler based compression [8] – after the BWT and MTF transformations are applied, a single stream of symbols needs to be coded, indicating the number of distinct symbols used since that token (which might again be a word) was last encountered. The gaps between adjacent document identifiers in the postings lists for some term appearing in the inverted index for a text collection, and the frequencies of occurrence of those terms, provide a third application of compression in which a single stream of symbols must be entropy-coded. Several of these scenarios are used as the basis for the experimentation described in Section 5.

2.2 Entropy Coding

We now provide details of the specific problem we consider in this paper.

Definitions and Terminology. Let $\sigma \in \Sigma^*$ be an m-sequence over the alphabet Σ, where $\Sigma = \{0, \ldots, n-1\}$ for some upper limit n, and with $\sigma_i \in \Sigma$ the i th element of σ, for $0 \le i < m$. Define $c(\sigma, s) = |\{i \mid 0 \le i < m \text{ and } \sigma_i = s\}|$, for each symbol $0 \le s < n$. Where there is no ambiguity we

Symbol	Explanation
n	Alphabet size
Σ	Source alphabet, $\Sigma = \{0 \ldots n-1\}$
σ	Sequence of symbols to be coded, $\sigma = \langle \sigma_i \in \Sigma \mid 0 \le i < m \rangle$
$c(\sigma, s)$	Number of occurrences of symbol s in σ
m	Length of σ and hence sum of symbol frequencies $c(\sigma, s)$
$\hat{c}(\sigma, s)$	Scaled frequency distribution derived from $c(\sigma, s)$
M	Adjusted sum of frequencies $\hat{c}(\sigma, s)$, usually a power of two

Table 1. Glossary of terminology used.

will use $c(s)$ as an abbreviation for $c(\sigma, s)$; that is, $c(s)$ is the frequency within σ of symbol s, and $m = \sum_{s=0}^{n-1} c(s)$. Table 1 summarizes these definitions.

Information Content. The *self-information* of σ, denoted $\mathcal{H}(\sigma)$ and measured in bits, is then defined by

$$\mathcal{H}(\sigma) = \sum_{s=0}^{n-1} c(\sigma, s) \log_2 \frac{m}{c(\sigma, s)}\,, \tag{1}$$

with $0 \cdot \log_2(m/0)$ taken to be zero. The self-information of a sequence is a measure of the minimum cost of representing that sequence as a bitstring if the symbols σ_i occurring in it are independent, and assuming that the n-element frequency distribution $c(\sigma, \cdot)$ is known to both encoder and decoder without any associated "cost" being incurred. That is, the self-information is computed assuming that a perfect code is available. It is also convenient to measure information content on a per-symbol basis; the *self-entropy* of a sequence σ is its self-information divided by its length: $\mathcal{H}(\sigma)/m$. The self-entropy can be thought of as being an attribute of the generation process, rather than of any particular message emitted by that mechanism.

If a code is used to represent a message σ, but is defined by a set of approximated symbol counts $\hat{c}(s)$ that sum to M rather than being defined by the symbol occurrence counts $c(\sigma, s)$, it is useful to compute the inefficiency that results, by assuming that an entropic code is constructed using the approximate probabilities, meaning that the message cost becomes:

$$\left(\sum_{s=0}^{n-1} c(\sigma, s) \log_2 \frac{M}{\hat{c}(s)} \right). \tag{2}$$

This is a value that is related to the Kullback-Leibler divergence of the two distributions, and is equal to $\mathcal{H}(\sigma)$ for perfectly matched approximate frequencies, for example, if $\hat{c}(s) = w \cdot c(\sigma, s)$ for some fixed constant w; and is greater than $\mathcal{H}(\sigma)$ if the approximate counts yield a different probability distribution. For example, if $c(\sigma) = \langle 3, 3, 2, 1, 1 \rangle$ with $m = 10$, then the self-information is 21.710 bits; and if the approximate distribution $\hat{c}(s) = \langle 5, 4, 3, 2, 2 \rangle$ is used instead, a perfect coder will generate an output that is 21.864 bits long when representing the same initial sequence of ten symbols. That is, the approximated symbol frequencies introduce a 0.713% compression loss.

Adding the Cost of the Model Parameters. In most practical compression systems, the distribution $c(\sigma, \cdot)$ cannot be assumed to be without cost, and the compressed message from a single-state semi-static compression system consists of two parts: a *prelude*, containing any necessary metadata such as the length m of the sequence σ, and the distribution of symbol frequencies $c(\sigma, s)$ or some approximation thereof; followed by the message *body*, containing the entropy-coded representation

of σ, based on knowledge of the contents of the prelude. The bound in Equation 1 relates to the minimum length of the message body, with the prelude an overhead to that cost. That is, there is tradeoff possible between the level of detail provided in the prelude in regard to symbol probabilities, and the cost incurred the message body when those (possibly approximated) probabilities are used. Overall, it might be cheaper to make use of approximate probabilities and somewhat inexact entropy codes than to use precise probabilities and exact entropy codes.

2.3 Huffman Coding

Three quite different entropy coding mechanisms have emerged: Huffman (or *minimum redundancy*) coding; *arithmetic* coding; and most recently, asymmetric numeral systems. The first and second of these three methods are considered in the remainder of this section. We start with Huffman coding.

Computing Codeword Lengths. David Huffman's algorithm [28, 32, 44] is known to most computing graduates. Given a set of symbol weights $c(s)$, it calculates a prefix-free code that minimizes the expected cost of coding an individual symbol from Σ, on the assumption that discrete codewords must be assigned, one per symbol $s \in \Sigma$. The assignment of codewords to symbols is defined in terms of an n-leaf binary code tree, with a symbol $s \in \Sigma$ labeling each leaf, and edges labeled with zeroes and ones. Moffat and Turpin [36] describe a range of efficient implementations of the underlying paradigm, tailored for specific coding instances; the key observation that they build on is that it is the *length* of the codewords that is the desired output, not the particular combinations of edge labels and codes. In particular, any set of integral codeword lengths ℓ_s for $0 \le s < n$ such that

$$\sum_{s=0}^{n-1} 2^{-\ell_s} \le 1 \tag{3}$$

can be converted into a set of prefix-free codewords; the cost of using that code is then given by

$$\sum_{s=0}^{n-1} \ell_s \cdot c(\sigma, s)\,. \tag{4}$$

Huffman's algorithm provides a mechanism for minimizing Equation 4 whilst remaining consistent (in the case of a binary channel alphabet, at equality) with Equation 3.

Implementations of Huffman's algorithm execute quickly, even for alphabet sizes into the millions, and the cost of computing a Huffman code is typically an inconsequential fraction of the cost of then using it in a second pass to code the input sequence.

Fast Decoding. In situations in which response time to data requests is important, decoding throughput is a key factor that affects usefulness. The flexibility afforded by computing codeword lengths rather than codewords allows systematic arrangements of codewords to be employed, minimizing the computational effort required at each decoding step. Moffat and Turpin [36] describe how *canonical Huffman codes* [10] can be rapidly processed, with each decoded symbol requiring (only) a small-scale search in an array of fewer than 32 elements, and then a shift and mask operation to bring in replacement bits from the incoming bitstream. It is also possible for larger amounts of memory to be used to build explicit decoding tables, so that multiple frequently-occurring symbols (with short codewords) can be decoded at each such cycle. We employ an implementation of canonical Huffman coding as one of the baseline techniques in the experiments described in Section 5. Moffat [32] provides a detailed description of Huffman coding and related mechanisms, including the use of canonical codes; and Alakuijala et al. [1] have recently invested considerable engineering effort into refining Huffman coding as a component of a complete compression system.

Prelude Representation. In a Huffman code, it is sufficient for the prelude to indicate which symbols $s \in \Sigma$ appear in σ (the *subalphabet*), together with the codeword length ℓ_s for each symbol. That is, the exact symbol frequencies are not required. Once the subalphabet and codeword lengths have been extracted, the decoder is able to reconstruct the same canonical code as was employed by the encoder. Given that the longest Huffman codeword is $n - 1 \approx n$ bits, it thus takes at most $n\lceil \log_2 n \rceil$ bits to store the n codeword lengths, versus (as many as) $n\lceil \log_2 m \rceil$ bits to store the n occurrence frequencies $c(s)$ using an equivalently straightforward approach. Turpin and Moffat [43] describe methods for further trimming the space required by the stored prelude, relative to these estimates; and Gagie et al. [20] describe compact structures for managing the code tables in memory during decoding operations.

2.4 Arithmetic Coding

General purpose multi-symbol-alphabet compression via arithmetic coding was described by Witten et al. [46], with refinements to that mechanism, and an evaluation of the data structures needed, provided eleven years later by Moffat et al. [33]. Howard and Vitter [25, 27] have also contributed to this early development. This subsection briefly summarizes that work.

Underlying Process. The basics of arithmetic coding are described in many textbooks [2]. The key idea is that of a current coding state, described by a $[L, L + R)$ tuple, where L is the current lower bound, and R the current range. At each coding step the existing interval is replaced by a (usually) narrower one, $[L', L' + R')$, where $L \le L'$ and $R' \le R - (L' - L)$, with the narrowing of the interval carried out in numeric proportion to the probability range of the next symbol within the $[0, 1)$ interval. In the case of probabilities based on occurrence frequencies $c(s)$ in σ, if $cumfreq[s] = \sum_{i=0}^{s-1} c(i)$ is the cumulative sum of symbols prior to symbol s in the alphabet, then the region of $[0, 1)$ allocated to s is given by $[cumfreq[s]/m, cumfreq[s + 1]/m)$. That is, at each coding step, $R'/R = c(s)/m$.

Prior to any symbols being coded, $L = 0$ and $R = 1$. After the complete sequence σ has been coded, and assuming that arbitrary-precision arithmetic is available, the final interval $[L, L + R)$ completely captures σ, and can be represented by any single value X that satisfies $L < X < L + R$. Such a value X cannot require more than $(-\log_2 R) + 2$ bits. Hence, because the final value of R is given by

$$\prod_{i=0}^{m-1} \frac{c(\sigma_i)}{m} = \prod_{s=0}^{n-1} \left(\frac{c(s)}{m}\right)^{c(s)} ,$$

arithmetic coding represents σ in at most $\mathcal{H}(\sigma) + 2$ bits.

Compared to Huffman coding, arithmetic coding offers two significant advantages: it handles skewed probability distributions accurately, in which one symbol s has $c(s)/m > 0.5$, representing occurrences of that symbol in less than one bit per instance; and it readily supports adaptive models, in which the probabilities and even contexts used for symbols change step by step. On the other hand, it decodes more slowly than does canonical Huffman coding, even when the symbol frequencies are fixed and the compression system is static or semi-static.

Incremental Encoding and Decoding. Practical implementations of arithmetic coding avoid arbitrary-precision arithmetic, and instead include a *renormalization* process that emits a bit (or byte), and doubles R (or multiplies it by 256), whenever a leading bit (or byte) of L can be unambiguously determined. That ability means that L and R can be manipulated as 32- or 64-bit integers [33, 46], and hence that coding one symbol involves a small number of multiplication and integer division operations to compute L' and R' from L, R, $cumfreq[s]$, and $c(s)$, followed by

zero or more renormalizations. The fact that the calculations are not exact means that some slight compression inefficiency emerges, but in practical systems it is very small.

Data Structures. When encoding, the required arithmetic is based on values that can be pre-computed and stored in tables of size n elements, and a fixed number of operations are required per input symbol. But when decoding, a target value t is first computed, and then the set of cumulative frequencies $cumfreq[s]$ must be searched, to identify the s for which $cumfreq[s] \le t < cumfreq[s+1]$. That search step means that for large-alphabet applications of the kind we consider here, care is required to ensure that the searching process is fast. In semi-static situations, $O(\log n)$ time is required for a binary search implementation; and it is also possible to carry out the same step in $O(1 + \log b)$ time, where b is the number of bits associated with the code for the symbol in question. Moffat and Turpin [36] provide details of these structures.

Prelude Representation. Semi-static arithmetic coding supposes that the set of n symbol frequencies $c(s)$ is available to the decoder, a more onerous requirement than the codeword lengths associated with Huffman coding, because of the increased precision in the numbers involved. That increased precision is what allows more precise probability estimates to be employed, and is what takes arithmetic coding closer to the self-information bound captured by $\mathcal{H}(\sigma)$, meaning that on all but short messages, the additional prelude cost (plus more) can be recouped.

2.5 Non-Entropy Codes

Generic integer codes have also been successfully employed in certain applications. For example, when the probability distribution over symbols is decreasing (that is, $c(s) \ge c(s+1)$) and hence symbol one is the most common, integer codes such as those of Golomb [22] and Elias [18] can sometimes be effective. Another range of approaches make use of a byte-based channel alphabet, typically providing faster decoding [5, 6, 12, 13, 42, 45] than bit-based codes, but with further compression degradation compared to the entropy codes that are the focus of this work.

3 ASYMMETRIC NUMERAL SYSTEMS

The third semi-static entropy coder we consider is a much more recent addition to the field, and is due to Jarosław (Jarek) Duda [15–17]. His *asymmetric numeral systems* mechanism (ANS) has connections to both Huffman coding and arithmetic coding, but is also distinct from both. This section describes two different ways of representing the ANS processes: a first intuitive description that allows the overall properties of ANS coding to be grasped; and then a second more detailed version that slightly alters the representations, but better supports practical implementation.

3.1 Underlying Process

The ANS process manipulates a single integer variable, the current *state*. In the simple form of ANS coding, *state* is initialized to zero, and thereby encodes the empty string. Each non-empty string over Σ is then assigned a unique corresponding positive integer, with *state* monotonically increasing as symbols are appended to make longer strings. The rate at which *state* grows as a symbol σ_i is appended is proportional to $m/c(\sigma_i)$. That is, whereas an arithmetic coder narrows down the range of an integer approximation of a real number with the high-significance bits converging in proportion to the symbol probability, ANS coding adds bits at the low-significance end of an integer value, with the *state* diverging from zero according to the reciprocal of the symbol probability.

Figure 1 provides an example of ANS coding. Suppose that Σ = {"a", "b", "c", "d", "e"} (using letters rather than integers to label the symbols, to avoid ambiguity), that c("a") = c("b") = 3, that c("c") = 2, and that c("d") = c("e") = 1, with $n = 5$ and $m = 10$. The five symbols are each

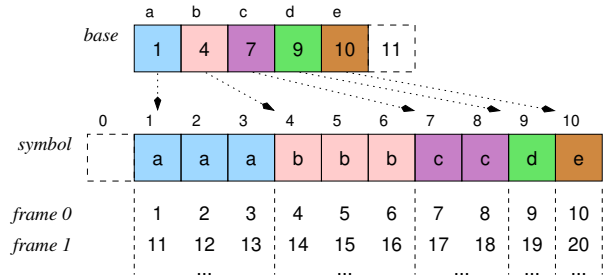

Fig. 1. Example of ANS symbol allocation across frames. In this example the source alphabet contains five symbols, "a", "b", "c", "d", and "e"; with $c(\text{"a"}) = c(\text{"b"}) = 3$, $c(\text{"c"}) = 2$, and $c(\text{"d"}) = c(\text{"e"}) = 1$; and hence frames of size $M = m = 10$ can be employed. The additional element in $base[\cdot]$ is a sentinel, equal to $base[n-1] + c(n-1)$. In this version there is also an additional value in $symbol[0]$ that is not used. A second slightly different arrangement is presented shortly.

		Current *state*															
		strt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	···
"a"	3/10	1	2	3	**11**	**12**	**13**	21	22	23	**31**	**32**	**33**	41	42	43	···
"b"	3/10	4	5	6	**14**	**15**	**16**	24	25	26	**34**	**35**	**36**	44	45	46	···
"c"	2/10	7	8	**17**	**18**	27	28	**37**	**38**	47	48	57	58	67	68	77	···
"d"	1/10	9	**19**	29	**39**	49	59	69	79	89	99	···					
"e"	1/10	10	**20**	30	**40**	50	60	70	80	90	···						

Table 2. Transition table $T(\cdot,\cdot)$ corresponding to the ANS coding arrangement shown in Figure 1. The $M = 10$ entries associated with the second frame are highlighted in blue; the $M = 10$ entries associated with the fourth frame are highlighted in red. The vertical lines in each row mark the end of each repetition of the frame.

allocated a range of indices in a *frame* of length M. In this example, M is taken to be m, the length of the message and hence also the denominator of the symbol probabilities; more generally, a value $M \geq m$ can be chosen, with adjusted counts scaled by M/m (and then rounded to integers) computed so as to result in any desired denominator M. In particular, having M a power of two allows fast decoding strategies to be employed, described shortly.

Once the frame template has been determined, the set of all positive integers is then laid out below the frame in rows of length M, with each integer x uniquely associated with a frame repetition number f (the row number) and an offset within the frame (the column position). For example, in Figure 1 the integer 14 is the fourth item in the second row; and, like all fourth items (and all fifth and sixth items too), is associated with the symbol "b". Based on that structure, Table 2 gives another view of the same ANS configuration. Now the integers associated with each symbol in Figure 1 are listed against that symbol, in the order that they appear through the set of frames. Considering Table 2 in detail, the regular structure means that the following patterns arise:

- Because it employs $M = 10$, each row of the table contains (only) numbers that are offset by the same amounts from multiples of 10 – for example, all of the values in the first row end in "1" or "2" or "3", and all of the values that end in "7" appear in the third row, corresponding to "c".
- Hence, given any value x, its location in the table can be directly computed using "mod" and "div" operations, based on knowledge of $c(s)$, how many (of the $M = 10$) different offsets occur

Initial → Sequence of *state* values	σ
50,001 → 15,000 → 1499 → 149 → 14 → 3 → 2 → 1 → 0	"aaabddea"
50,002 → 15,001 → 4500 → 449 → 44 → 12 → 4 → 0	"babdeaa"
50,003 → 15,002 → 4501 → 1350 → 134 → 39 → 3 → 2 → 1 → 0	"aaadbeaaa"
50,004 → 15,000 → 1499 → 149 → 14 → 3 → 2 → 1 → 0	"aaabddeb"
50,005 → 15,001 → 4500 → 449 → 44 → 12 → 4 → 0	"babdeab"
50,006 → 15,002 → 4501 → 1350 → 134 → 39 → 3 → 2 → 1 → 0	"aaadbeaab"
50,007 → 10,000 → 999 → 99 → 9 → 0	"dddec"
50,008 → 10,001 → 3000 → 299 → 29 → 2 → 1 → 0	"aaddeac"
50,009 → 5000 → 499 → 49 → 4 → 0	"bdded"
50,010 → 5000 → 499 → 49 → 4 → 0	"bddee"

Table 3. ANS decoding of integers relative to the example configuration shown in Figure 1 and Table 2, covering the ten integers that make up the 5000 th frame. Note the pattern of final symbols in the decoded strings σ, matching the underlying frame arrangement. The prefix sequences prior to that final symbol also exhibit patterns.

in each symbol's row, and knowledge of which particular set of $c(s)$ offsets are in use in each row.

- Similarly, the exact value of the i th entry in each row can be computed as a simple function of i, M, and the set of $c(s)$ values.
- More generally, that i th entry in each row is approximately $i \cdot M/c(s)$.

That is, Table 2 defines a mapping $T(state, s)$ that associates a distinct integer with every combination of *state* and symbol s; moreover, $T(state, s)$ is, by construction, strictly greater than *state*. To encode a sequence of symbols, say σ = "abaebbdcac" (a string that matches the counts $c(\cdot)$ used in Figure 1 and Table 2) a variable *state* is initialized to zero, and then the iterator $state' \leftarrow T(state, \sigma_i)$ is applied for each $\sigma_i \in \sigma$. Looking at Table 2, $T(0,$ "a") is 1; then $T(1,$ "b") is 5; and so on. The complete sequence of states traversed for the example string σ = "abaebbdcac" is $0 \rightarrow 1 \rightarrow 5 \rightarrow 13 \rightarrow 140 \rightarrow 466 \rightarrow 1555 \rightarrow 15{,}559 \rightarrow 77{,}798 \rightarrow 259{,}323 \rightarrow 1{,}296{,}618$. That final value requires 21 bits as a binary number, assuming that the decoder somehow knows how long the encoded form is, and also knows the frame layout. In contrast, a Huffman code for the same set of five symbol occurrence counts would assign three two-bit codewords (symbols "a", "b", and "c"); and two three-bit codewords (symbols "d", "e"), with a total cost of 22 bits for the example string. By using exact probabilities rather than approximate probabilities adjusted to negative powers of two (which is what implicitly occurs when a Huffman code is constructed), the ANS coder is able to attain superior compression, even on this non-extreme distribution. Note, however, that the superiority is not guaranteed: the reverse sorted string "edccbbbaaa", which shares the same mix of symbol probabilities as "abaebbdcac", corresponds to a final ANS state of 3,768,342, and requires 22 bits as a binary value; and the forwards sorted string "aaabbbccde" yields a final state of 389,800 and consumes only 19 bits. The range of output lengths illustrated by these two extreme examples is addressed when the second version of ANS is introduced in the next section.

Decoding requires that the transformation be reversed via the iterator $(state', \sigma_i) \leftarrow T^{-1}(state)$, starting with a value of *state* that represents the whole of the input sequence, and one-by-one generating the symbols that it contains, extracting them in reverse order. Table 3 gives further examples of correspondence between strings and integers using the example $M = 10$ frame shown in Figure 1 and Table 2, based on the counts $c(\cdot) = \langle 3, 3, 2, 1, 1 \rangle$. The ten strings shown cover the full span of the 5000 th frame.

```
1: function ANS_encode(σ_{0...m−1}):
2: state ← 0
3: for i ← 0 to m − 1 do
4:     f ← state div c(σ_i)
5:     r ← state mod c(σ_i)
6:     state ← f · M + base[σ_i] + r
7: return state
```

```
1: function ANS_decode(state):
2: σ ← ∅ and m ← 0
3: while state > 0 do
4:     r ← 1 + (state − 1) mod M
5:     f ← (state − r) div M
6:     σ_m ← symbol[r]
7:     state ← f · c(σ_m) + (r − base[σ_m])
8:     m ← m + 1
9: return reverse(σ_{0...m−1})
```

Fig. 2. Transforming a sequence into a unique integer, and reversing that transformation, where $c(s)$ is the number of instances of symbol s in each frame of size M; where $base[s]$ is the first instance of s in each frame; and $symbol[r]$ is the symbol associated with the r th position in each frame (see Figure 1). Note that $c(s) = base[s+1] - base[s]$, and that $base[\cdot]$ is the only array required in the encoder. This initial implementation assumes infinite-precision integer arithmetic, frame offsets that run from 1 to M, as illustrated in Figure 1 and Table 2, and contiguous symbol allocations within the frame (the rANS arrangement).

3.2 Range ANS Coding

In the approach shown in Figure 1 and Table 2 – denoted as *range ANS coding*, rANS, by Duda [15–17] – all $c(s)$ instances of symbol s are retained as a contiguous block in each frame, introducing (as was noted by the two extreme examples) a slight bias that favors symbols that appear early in the frame. The alternative is to distribute the $c(s)$ instances of s uniformly through the frame; we discuss such an arrangement shortly.

The range ANS coder's contiguous symbol blocks mean that encoding and decoding can be carried out using integer arithmetic and either one $n+1$-element array (in the encoder), or one $n+1$-element array plus one $M+1$-element array (in the decoder). Figure 2 gives details of the computation of the forwards ANS mapping and inverse mapping, in both cases assuming that the arrays $base[0 \ldots n]$ and $symbol[0 \ldots M-1]$ have been precomputed. The critical arithmetic that implements $T(\cdot,\cdot)$ and $T^{-1}(\cdot)$ appears at steps 4 to 6 in the encoder, and at steps 4 to 7 in the decoder.

Only $base[\cdot]$ is required during the forwards mapping, since $c(s) = base[s+1] - base[s]$. The second – and potentially much larger – array $symbol[\cdot]$ is added in the decoder, so that offsets r within the frame can be converted to symbol identifiers via direct table lookup in $O(1)$ time. In the rANS coder the array $symbol[\cdot]$ could be dispensed with if a binary or linear search in $base[\cdot]$ was used to compute the same mapping, but at the cost of $O(\log n)$ time per symbol decoded. Note that the same mechanism can also be applied in arithmetic coding – the $O(\log n)$-time search in the $cumfreq[\cdot]$ array to find the decode *target* could be avoided if an M-element lookup table was constructed and made available.

3.3 Optimality

If it is assumed that arbitrary-precision arithmetic is employed, then Figure 2 and the discussion in connection with Table 2 provide an explanation as to why ANS performs well as an entropy coder: as each symbol σ_i is encoded, the new i th value of *state*, denoted $state_i$, is computed from the $(i-1)$ th value $state_{i-1}$ such that

$$\lim_{i \to \infty} \frac{state_i}{state_{i-1}} = \frac{M}{c(\sigma_i)} \,.$$

Hence, over a long sequence σ, and assuming that the number of bits required to represent the final state is logarithmic in its value, compression effectiveness is given by

$$\log_2 state_{m-1} \approx \log_2 \left(\prod_{i=0}^{m-1} \frac{M}{c(\sigma_i)} \right) \approx \sum_{s=0}^{n-1} \left(c(s) \cdot \log_2 \frac{M}{c(s)} \right) \approx \mathcal{H}(\sigma)$$

provided that M is taken to be m, and that the frame is constructed using the observed frequencies $c(\sigma, s)$. In practice there are a number of issues that affect this ideal bound, including the overhead cost of indicating the number of bits required to represent *state* (of the order of $\log_2 \log_2 state_{m-1}$), the slight effects associated with the mod and div operations when i is small, and the prelude cost associated with conveying the counts $c(s)$ to the decoder so that it can construct the same frame. Duda [15, 16] considers some of these issues.

In situations where one symbol dominates, ANS shares arithmetic coding's ability to represent a sequence in less than one bit per symbol. For example, the 21-sequence σ = "aaaabcaaaaaaabaaaaaaa" with $c(\cdot) = \langle 18, 2, 1 \rangle$ leads to a final state of 118,232, and requires 17 bits to represent 21 symbols.

3.4 Incremental Encoding and Decoding

ANS coding would be of only limited value if arbitrary-precision integer arithmetic were required. Fortunately, mechanisms have been developed that allow periodic renormalization steps, and the use of finite-precision arithmetic. In arithmetic coding, the bits emitted are "confirmed" bits at the most-significant end of the computation. In ANS, the bits emitted come from the least-significant end of the computation, with no sense of them being "confirmed" in any way. Surprising as this may sound, it is nevertheless perfectly feasible.

Before presenting the details of the renormalization mechanism, we first introduce several changes to the illustrative approach that was shown in Figure 1, Table 2, Table 3, and Figure 2. First, we number the symbol offsets within each frame so that they are labeled from 0 to $M - 1$, rather than the previous 1 to M arrangement. This means that there is no longer an integer code reserved for the empty string, and also that care must be taken when initializing *state*, since fixed-points can (and do) arise for which $x = T(x, s)$.

Second, we allow any permutation of the symbols within each frame, and require only that there are $c(s)$ occurrences of s within the M symbols comprising the frame. That change is accomplished by introducing a permutation vector $shuffle[0 \ldots M-1]$ that can be either a random assignment of the frame's symbols, or a systematic layout of them that seeks to uniformly spread the $c(s)$ occurrences of s [14]. This change, and the use of 0-origin frame numbering, are illustrated in the top two arrays depicted in Figure 3. In tandem with $shuffle[\cdot]$, a second new array $rank[\cdot]$ is introduced, to allow the relative ordering between instances of each symbol s to be correctly maintained. As well, for reasons that will become apparent shortly, it is desirable to use a frame of $M = 16$ items. To achieve that, the previous $M = 10$ symbol frequency distribution used in Figure 1 has been scaled to create a new approximate distribution $\hat{c}(\cdot) = \langle 5, 4, 3, 2, 2 \rangle$. The scaling process is discussed in more detail below.

Third, we introduce the quantity *radix*, the channel alphabet size. For binary representations, where the output unit is the bit, $radix = 2$. But for some applications there may be advantages in using $radix = 256$ and generating output units that are bytes, or even $radix = 65{,}536$ and output units that are 16-bit short integers, noting that bytes and shorts are almost certainly faster to process than are bits.

Finally, bounds are introduced on the values of *state* that can be allowed. The lower bound is given by *sml*, which is computed as the product of M and K, where K is a positive integer that controls the magnitude of the arithmetic that is required, and adjusts the number of bits of precision

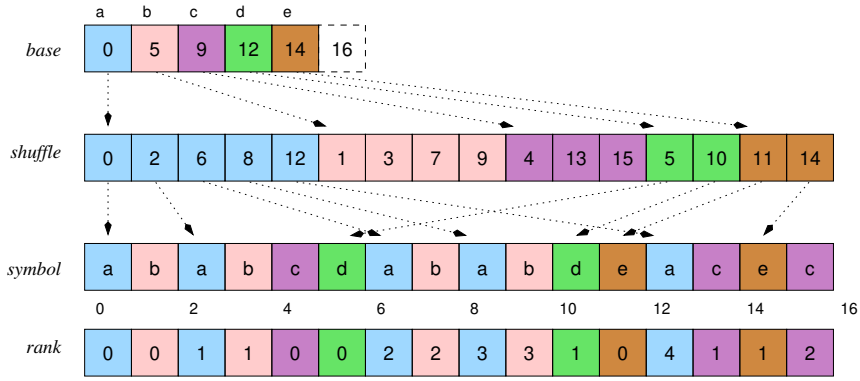

Fig. 3. Example of ANS symbol allocation across frames using a shuffled symbol ordering and scaled frequencies, here with $M = 16$ and $c(\text{“a”}) = 5$, $c(\text{“b”}) = 4$, $c(\text{“c”}) = 3$, and $c(\text{“d”}) = c(\text{“e”}) = 2$, as an approximation of the frequency distribution $\langle 3, 3, 2, 1, 1\rangle$ shown in Figure 1. The encoder requires the n-array $base[\cdot]$ and the M-array $shuffle[\cdot]$, which records the permutation being used; the decoder requires $base[\cdot]$ (to compute $c(\cdot)$), the M-array $symbol[\cdot]$, and the added partial inverse permutation M-array $rank[\cdot]$, with $rank[i]$ counting the number of instances of $symbol[i]$ that appear in $symbol[0 \ldots i-1]$.

maintained in intermediate values. The upper bound is a factor of *radix* larger:

$$sml \le state < sml \cdot radix\,.$$

In order to establish that invariant, *initial_state* must be a value in the range *sml* to $sml \cdot radix - 1$. One standard choice is simply to take *sml*, but it is worth noting that the range that is available means that the value chosen can be used to encode other information, perhaps even (via an application of the process shown in Figure 2) whatever length prefix of σ can be fitted into the available bits. To maintain the invariant on an ongoing basis, each time an encoding step arises that would, were it allowed to proceed, take *state* beyond the upper end of the allowed range, a pre-emptive *renormalization* step is carried out to reduce *state*, so that following that next encoding step, *state* arrives back into the required range.

Figure 4 provides details of the renormalization process. As before, it is assumed that a set of symbol counts $c(\cdot)$ that sum to the frame size M are supplied, and that the arrays $base[\cdot]$ and $shuffle[\cdot]$ (encoder), or $base[\cdot]$, $symbol[\cdot]$ and $rank[\cdot]$ (decoder) have been pre-computed, based on the counts $c(\cdot)$. Those counts might have been derived from the string σ and hence be an exact match for its probability distribution; might be approximations of the exact counts in order to allow M to be a power of two (the situation assumed for the purposes of Figure 4 and Table 4, discussed shortly); or might be static and not include any knowledge of σ.

The critical part of the encoding function *ANS_encode*$(\cdot)$ shown in Figure 4 is the loop at steps 7 to 10. It starts with a *state* value that is within the required range, and, with the knowledge that symbol σ_i is about to be encoded, repeatedly extracts low-order digits out of *state* (each a value in the range 0 to $radix - 1$), continuing until a new reduced *state* value is arrived at from which coding σ_i is certain to take *state* back into the target range. The boundary value that defines the maximum “safe” starting point from which to encode σ_i is defined by $K \cdot radix \cdot c(\sigma_i) - 1$. That is, the bound to encode a symbol σ_i is a simple function of $c(\sigma_i)$ and the frame parameters, and can be precomputed for each symbol $s \in \Sigma$ and stored in an n-element array.

The transition table shown in Table 4 helps explain this process. In the table the boxed blue values – one complete frame – are the ones that represent “safe” ending points for state transitions,

```
1: function ANS_encode(σ_{0...m−1}):
2: sml ← K · M                          ▷ lower bound on range
3: state ← initial_state
4: b ← 0                                ▷ output counter
5: for i ← 0 to m − 1 do
6:     assert: sml ≤ state < sml · radix
7:     while state ≥ K · radix · c(σ_i) do
8:         buffer[b] ← state mod radix
9:         b ← b + 1
10:        state ← state div radix
11:    f ← state div c(σ_i)
12:    r ← state mod c(σ_i)
13:    state ← f · M + shuffle[base[σ_i] + r]
14: encode(buffer, b, state − sml)
15: encode(buffer, b, m)
16: return ⟨buffer, b⟩
```

```
1: function ANS_decode(buffer, b):
2: sml ← K · M
3: m ← decode(buffer, b)
4: state ← decode(buffer, b) + sml
5: for i ← 0 to m − 1 do
6:     assert: sml ≤ state < sml · radix
7:     f ← state div M
8:     r ← state mod M
9:     σ_i ← symbol[r]
10:    state ← f · c(σ_i) + rank[r]
11:    while state < sml do
12:        b ← b − 1
13:        state ← state · radix + buffer[b]
14: assert: state = initial_state
15: return reverse(σ_{0...m−1})
```

Fig. 4. Complete ANS encoding and decoding processes, including renormalization for *state* and incremental output (encoder) and input (decoder). The auxiliary function *encode*(*buffer*, *b*, *val*) uses a non-ANS mechanism (for example, binary for *state* − *sml*, and the Elias δ code for *m*) to append *val* to *buffer*[·] and update the output pointer *b*; similarly, *decode*(*buffer*, *b*) extracts and returns that value from the encoded message in *buffer*[·]. Note also that it is assumed that the decoder processes *buffer*[·] from right to left.

		Current *state*, counting from zero											
		0	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	10	···
“a”	5/16	0	2	6	8	12	**16**	**18**	**22**	**24**	**28**	32	···
“b”	4/16	1	3	7	9	**17**	**19**	**23**	**25**	33	···		
“c”	3/16	4	13	15	**20**	**29**	**31**	36	···				
“d”	2/16	5	10	**21**	**26**	37	···						
“e”	2/16	11	14	**27**	**30**	43	···						

Table 4. Revised state transition table $T(\cdot,\cdot)$, using a scaled frequency distribution with c(“a”) = 5, c(“b”) = 4, c(“c”) = 3, and c(“d”) = c(“e”) = 2, and a shuffled frame of size $M = 16$ in which the symbol ordering is “ababcdababdeacec” (see Figure 3). In this example $K = 1$ and *radix* = 2, and hence *sml* = 16 and the required invariant is $16 \le state < 32$.

safe because the mapped values fall in the desired range. All other transitions either undershoot (to the left of the enclosed area, with final state values less than 16) or overshoot (to the right of the enclosed area, with final state values greater than 31), and hence cannot be countenanced. That then limits the valid “pre-range” for each transition to a subset of 2 . . . 9, with the exact pre-range dependent on the symbol giving rise to the transition.

As an example, if the next symbol to be coded is an “a”, then $K \cdot radix \cdot c(\sigma_i) = 1 \cdot 2 \cdot 5 = 10$, and the previous values of *state* for which $T(state,$ “a”$)$ returns into the range 16 . . . 31 are 5 . . . 9 inclusive. Hence, if *state* is currently 16, and an “a” is about to be coded, then one renormalization iteration suffices, with a 0-bit generated to the output, and with *state* halved to 8. The coding of the “a” then takes *state* up to 24, a value that is within the required final range of 16 . . . 31. On the other hand,

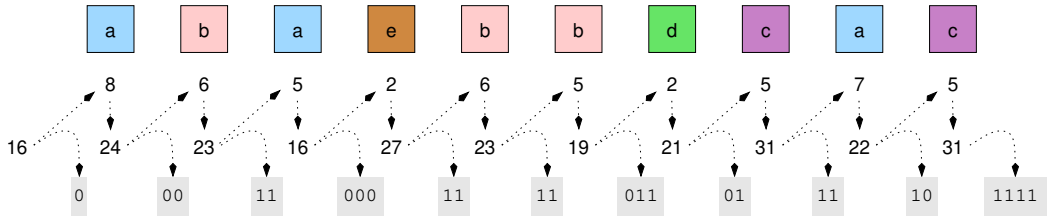

Fig. 5. Tracing the encoding process shown in Figure 4 for the input string "abaebbdcac", starting with *initial_state* = 16, and assuming that the final state of 31 in the range 16–31 is represented in four bits. The additional bits required to represent m are not included in this example. The decoding process follows exactly the same path in reverse, starting at the right of the diagram.

if *state* is initially 23 and the upcoming symbol is an "a", then two digits are emitted, a 1-bit that takes *state* to 11 (still outside the desired pre-range of 5 . . . 9), and then a second 1-bit, that takes *state* down to 5. The subsequent coding of the "a" then takes *state* to 16, within the required span of values at the conclusion of a coding step. Figure 5 includes both of these particular scenarios, as well as the other renormalization operations required to encode the sequence "abaebbdcac" starting at an *initial_state* of 16. Once all $m = 10$ symbols have been processed, a total of 21 bits have been generated via the renormalization loop, and a further 4 bits are required to represent the final value of *state* within the range $sml \ldots sml \cdot radix - 1$.

The decoder, shown in the right-hand side of Figure 4, exactly reverses the path taken by the encoder. It maintains the same range for *state*, starting with the value read from the compressed file at step 4, which means that after each symbol has been decoded, digits must be read from the compressed stream (in the array *buffer*$[\cdot]$) and appended at the right of *state*, as shown at step 13. In Figure 5, the decoding operation can be traced from right to left, simply reversing the orientation of each of the arrows. At each instant that a digit (in the figure, with $radix = 2$, a bit) is required, to be folded back into *state*, it is exactly the one that was emitted at the corresponding moment in the encoder's operation when considered in reverse.

The total of 25 bits that are generated for the example shown in Figure 5 is four more than the 21 bits that was noted for the same sequence in connection with Table 2 and Figure 2. The main reason for the increase is that *state* is assigned an initial value of *sml*, which itself contains four bits of content. That starting point shows up as additional cost through the course of the message. But as already noted, any *initial_state* within the range 16 . . . 31 (when $M = 16$, $K = 1$, and $radix = 2$) can be employed. That is, while the four bits do need to be spent, they can be used constructively if compression effectiveness is the over-riding goal. Note also that the encoder and decoder need to agree on how to represent the final value of *state* (step 14 in the encoder, and step 4 in the decoder), with binary the obvious choice; and how to represent the length $m = |\sigma|$ (steps 15 and 3 respectively), with suitable choices including the Elias δ code.

3.5 Table-Based Decoding

The mechanism illustrated in Figure 4 can be further streamlined. One option for fast decoding is to note that (unlike the situation with arithmetic coding) both of the key integer division operations at steps 7 and 8 in function *ANS_decode*$(\cdot, \cdot)$ involve a fixed value M that does not alter as decoding takes place. If M is a power of two, then those operations can be effected using shift/mask operations on the integer value *state*.

	Current *state*															
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$table[\cdot].sym$	a	b	a	b	c	d	a	b	a	b	d	e	a	c	e	c
$table[\cdot].newst$	5	4	6	5	3	2	7	6	8	7	3	2	9	4	3	5
$table[\cdot].nbits$	2	2	2	2	3	3	2	2	1	2	3	3	1	2	3	2

Table 5. Table-based decoding, with $M = 16$, $K = 1$, and hence $sml = 16$. Figure 6 shows how it is used.

1: // One-from-many table-based decoding, replacing steps 5 to 13 of Figure 4
2: **for** $i \leftarrow 0$ **to** $m - 1$ **do**
3: $\quad \sigma_i \leftarrow table[state].sym$
4: $\quad state \leftarrow shift\_and\_add\_bits(table[state].newst, table[state].nbits)$

Fig. 6. Table-based decoding loop for "one-from-many" operation, making use of a pre-computed table of the form shown in Table 5. Other decoding elements, including initialization of *state* and *m*, are as noted in Figure 4. The function *shift_and_add_bits*(*state*, *nbits*) is assumed to access the array *buffer* and the counter *b*, and bring *nbits* additional digits into the low-order positions of *state*.

Another possibility is to pre-compute the loop from step 5 to step 13 for each possible value of *state* in the range $sml \ldots sml \cdot radix - 1$, and store those numbers in a table. Table 5 shows the values that result for the example $M = 16$ frame shown in Figure 3 and Table 4, and Figure 6 shows how the pre-computed decoding table is used to accelerate the decoding and eliminate all of the decoding conditionals.

For this mechanism to be viable, the frame structure must be such that each current *state* value in the set of reachable ones corresponds to a set of "expanded" ranges that either completely lies within the $sml \ldots sml \cdot radix - 1$ interval, or completely outside it, regardless of what value the added digits have when folded in. For example, looking again at Table 4, the reachable states are the ones shown in blue, and so the set of expanded ranges from $2 \ldots 9$ are of interest. When $state = 9$, the expanded ranges are the intervals $18 \ldots 19$, $36 \ldots 39$, and so on; when $state = 8$ the expanded ranges are $16 \ldots 17$, $32 \ldots 35$, and so on; and when (at the lower end of the reachable set) $state = 2$ the expanded ranges are $4 \ldots 5$, $8 \ldots 11$, $16 \ldots 23$, $32 \ldots 47$, and so on; that is, in each case shifting left by one bit and considering all binary values that can be formed as a result. There are no cases amongst all these options in which the expanded range straddles the required interval for *state*, and hence every reachable state has a single unique value for *nbits*, the number of renormalization steps required, regardless of the actual digits being processed.

A sufficient condition for uniqueness to occur is if *sml* is an integer multiple of the greatest power of *radix* less than or equal to it. For example, if $radix = 5$, then suitable values of *sml* include 10, 15, 20; 25, 50, 75, 100; 125, 250, 375, 500; and so on. In particular, in the usual case in which $radix = 2$, uniqueness is assured by the sufficient condition that *sml* be a power of two, which in turn implies that both K and M are also powers of two. If for some reason it is impossible to make M a power of two, then other binary-aligned values for *radix* can still be considered. For example, if $radix = 16$, then viable values for *sml* include all of 16, 32, ..., 240; and then all of 256, 512, 768, ..., 3840; and then 4096, 8192 and so on. Then, with $sml = 3840$ say, $M = 1280$ and $K = 3$ becomes a valid combination, as does $M = 768$ and $K = 5$.

The decoding process described in Figure 6 and Table 5 provides for a "one-from-many" relationship between generated output symbols and decoded digits. That relationship arises as a

	Current *state*														
	16	17	18	···	126	127	128	129	130	···	251	252	253	254	255
table[·].*syms*	a--	b--	a--		e--	cb-	aa-	ba-	ab-		ee-	ac-	ce-	ec-	cc-
table[·].*nsyms*	1	1	1		1	2	2	2	2		2	2	2	2	2
table[·].*newst*	5	4	6		15	6	13	10	11		3	14	5	5	8

Table 6. Table-based decoding, with $M = 16$, $K = 1$, $radix = 16$, and hence $sml \cdot radix = 256$, and with "-" used to indicate "don't care" symbols. As many as $max\_decode\_syms = 3$ output symbols are generated prior to each input digit being read. For example, the state chain $160 \rightarrow 50 \rightarrow 16 \rightarrow 5$ gives rise to "aaa"; and the state chain $241 \rightarrow 60 \rightarrow 19 \rightarrow 5$ gives rise to "bab". Figure 7 shows the way this table is used.

```
1: // Many-from-one table-based decoding, replacing steps 5 to 13 of Figure 4
2: i ← 0
3: while i < m do
4:     copy max_decode_syms symbols from table[state].syms to σ_{i,i+1,...}
5:     i ← i + table[state].nsyms
6:     b ← b − 1
7:     state ← table[state].newst · radix + buffer[b]
```

Fig. 7. Table-based decoding loop for "many-from-one" operation, making use of a pre-computed table of the form shown in Table 6. In an implementation the values stored in *table*[*state*].*newst* would all be pre-multiplied by *radix*, to save that operation being required.

result of $\log_2 radix = 1$ being less than even the smallest of the five nominal codeword lengths in bits, with $\log_2(10/3) \approx 1.737$. When *radix* is large relative to the anticipated code lengths, the relationship reverses, and an alternative "many-from-one" arrangement is possible. For example, when $radix = 16$ and the renormalization is via "digits" that are four-bit nibbles between 0 and 15, for the frame $\hat{c}(\cdot) = \langle 5, 4, 3, 2, 2\rangle$ no renormalization step requires input of more than a single digit, and each input digit can account for as many as three output symbols.

The many-from-one relationship then allows a second form of table-based decoding. Table 6 shows three extracts from the corresponding $radix = 16$ decode table for the same example distribution $\hat{c}(\cdot)$ as used in Table 5; and Figure 7 shows how the decoding loop is refactored so that the iteration is over compressed digits rather than over decompressed symbols. An important part of Figure 7 is pre-computation of *max_decode_syms*, the maximum number of symbols that can derived from any state in the range $sml \le state < sml \cdot radix$. In the example table, $max\_decode\_syms = 3$. That value is then used to drive the fixed-length "copy symbols" operation at step 4. Rather than execute a second loop that copies symbols one by one, it is faster to copy a fixed block of symbols, including any trailing don't care values that may be required, shown in Table 6 as "-" elements. Following the fixed copy, the output pointer is incremented by the right number of symbols, so that any don't cares will be covered by the next copy. The observation that a "fixed-length copy" approach is more efficient that executing a loop that iterates the correct number of times is due to Martinez et al. [31], and has also been exploited by Pibiri et al. [39].

In both forms of table-based decoding, the variable quantity might be zero in some columns of the table. In the one-from-many arrangement (illustrated in Table 5), it might be that even after a symbol has been decoded, *state* remains within the required interval, in which case $table[state].nbits = 0$. Similarly, there may be configurations in the many-from-one table (illustrated in Table 6) from

```
1: function scale_frequencies(c(·), m, M):
2: m′ ← m and M′ ← M
3: for s ← n − 1 downto 0 do
4:     scale ← (s/n) · (M/m) + ((n − s)/n) · (M′/m′)
5:     ĉ(s) ← max(1, ⌊0.5 + scale · c(s)⌋)
6:     m′ ← m′ − c(s) and M′ ← M′ − ĉ(s)
7: return ĉ(·)
```

Fig. 8. Scaling the observed frequency distribution $c(\cdot)$ to reach a new total $M \geq n$ as the sum of an approximate frequency distribution $\hat{c}(\cdot)$. In the final iteration, when $s = 0$ and the most frequent symbol is being processed, the scale factor will be $M'/c(0)$, and $\hat{c}(0)$ will be set to the remaining unallocated count M', to ensure that the target M is reached.

which no symbol can be emitted, because one input digit was insufficient to boost *state* back into the required $sml \ldots sml \cdot radix - 1$ interval. If so, then there will again be columns in the table in which $table[state].nsyms = 0$.

The improved speed of table-based decoding comes at a cost in terms of memory space. The arrays associated with Figure 4 requires $n + 2M$ words of memory, regardless of K or *radix*. For typical character-based compression scenarios with $n = 256$ and (say) $M = 4096$, the total space required is thus small, and the decoding structures will easily fit into L1 cache and hence be rapidly accessible. On the other hand, the table structures assumed in Figure 6 and Figure 7 require $3K \cdot M \cdot (radix - 1)$ elements, and are no longer independent of K or *radix*. If $radix = 2$ and $K = 4$ is used with one-from-many decoding, then the same character-based scenario will require four times as much space, and is perhaps manageable. But if many-from-one decoding is to be pursued (as illustrated in Table 6) then *radix* must of necessity be large. If $radix = 256$ and $M = 4096$, then even with $K = 1$, more than three million values must be stored, already assuming that a string of *max_decode_syms* source-alphabet symbols can be stored as a single value. In this final case the speed gains achieved by the table-based approach might be partially or even completely eroded by the overhead cost of cache misses.

Najmabadi et al. [37] explore table-based ANS in more detail.

3.6 Scaled Distributions

Given a frequency distribution $c(s)$ that sums to m, it is thus desirable to approximate it with a scaled one $\hat{c}(s)$ that is as close as possible to $c(s)$ in terms of implied probabilities, but sums to a new value M. Equation 2 described how to compute the resultant cross-entropy cost; clearly, the closer the new probability distribution is to the starting one, the less compression loss will occur.

Figure 8 shows how this adjustment is carried out. Two ratios are blended before each frequency $c(s)$ is multiplied: a static ratio M/m, which reflects the initial objective; and an adaptive ratio, M'/m', which reflects the currently unallocated portion. At the beginning, when low-frequency symbols are being processed, the weight favors the static ratio; towards the end, as the dominant symbols are processed, the weighting favors the adaptive ratio. Note also that none of the $\hat{c}(s)$ values can be allowed to become less than one, a constraint that becomes necessary in cases in which $n \leq M < m$.

The scaled $M = 16$ frequency distribution $\hat{c}(\cdot) = \langle 5, 4, 3, 2, 2 \rangle$ used in Table 4 was computed by this approach. It has a cross-entropy of 2.186 bits per symbol and is approximately 0.713% inefficient relative to the $m = 10$ symbol distribution shown in Table 2. Similarly, a re-weighting to achieve $M = 12$ would result in $\hat{c}(\cdot) = \langle 4, 4, 2, 1, 1 \rangle$, a cross-entropy of 2.185 bits per symbol, and 0.645%

Arrangement	Frame	M	$K = 1$	$K = 2$	$K = 4$	$K = 16$
range	$c(\cdot) = \langle 3, 3, 2, 1, 1\rangle$	10	2.220	2.184	2.173	2.170
	$c(\cdot) = \langle 9, 9, 6, 3, 3\rangle$	30	2.220	2.184	2.173	2.170
	$\hat{c}(\cdot) = \langle 5, 4, 3, 2, 2\rangle$	16	2.210	2.186	2.186	2.185
	$\hat{c}(\cdot) = \langle 10, 10, 6, 3, 3\rangle$	32	2.231	2.190	2.177	2.173
shuffled	$c(\cdot) = \langle 3, 3, 2, 1, 1\rangle$	10	2.194	2.177	2.171	2.170
	$c(\cdot) = \langle 9, 9, 6, 3, 3\rangle$	30	2.173	2.171	2.170	2.170
	$\hat{c}(\cdot) = \langle 5, 4, 3, 2, 2\rangle$	16	2.186	2.183	2.184	2.185
	$\hat{c}(\cdot) = \langle 10, 10, 6, 3, 3\rangle$	32	2.174	2.172	2.172	2.172

Table 7. The effect of K on the compression effectiveness of ANS implementations, with compression rates measured in bits per symbol, for a single test file of $m = 1{,}000{,}000$ independent random integers in the range $0 \ldots 4$ generated according to the probability distribution $\langle 0.3, 0.3, 0.2, 0.1, 0.1\rangle$. Two different frame layouts and four different frame sizes are explored, using *radix* $= 2$ in all cases. The self-information of the input sequence is 2.170 bits per symbol.

inefficiency; and a target of $M = 25$ gives rise to an approximate distribution of $\hat{c}(\cdot) = \langle 8, 7, 5, 3, 2\rangle$, a cross-entropy of 2.179 bits per symbol, and a relative inefficiency of 0.360%. Note that the cross-entropy need not strictly improve as M is incremented, but that it will be non-increasing across multiples of any given value of M, through the sequence M, $2M$, $3M$, and so on.

3.7 Compression Effectiveness

Notable in Table 5 is that some instances of symbol "a" are followed by the input of one bit, and some are followed by the input of two bits; likewise, some instances of "c" are followed by two bits of renormalization, and some by three. It is this variability across the span of the frame that allows ANS to represent symbols in a bit-fractional manner. In the context of the scaled counts used in Table 5, the exact entropic cost of an "a" is $-\log_2(5/16) = 1.678$ bits, and by sometimes being associated with one bit, and sometimes with two, that fractional value is approximated across the instances of "a" in σ. The fidelity of the approximation is then determined by the number of different "a"-generating states that are available. When *sml* is insufficient, only a few states are in use and the quality of the approximation might be low. In Table 4, for example, the state interval is from $16 \ldots 31$ and there are just five "a"-generating states. Moreover, all of the state transitions are funneled through just eight intermediate points (Table 4), states $2 \ldots 9$.

The constant K that was introduced in Figure 4 helps address this issue. Choosing a value of K greater than one enlarges the state interval and adds repetitions of the frame, and hence provides a larger set of *state* options and a more fine-grained approximation to the entropic code lengths. For example, if $K = 4$, then the state interval becomes $64 \ldots 127$, and includes four repetitions of the basic 16-element frame. Table 7 explores the effect of varying K. To create the table, a sequence of 1,000,000 random values in $0 \ldots 4$ was generated, based on the probability vector $\langle 0.3, 0.3, 0.2, 0.1, 0.1\rangle$. That sequence was then coded using two different ANS versions covering contiguous frames (the rANS approach) and randomly shuffled frames; using four different values of K; and using four different values of M, two that accurately reflect the underlying probability distribution used to generate the test sequence, and two approximate ones that were created via the scaling process shown in Figure 8. All of the numbers in the table are measured compression rates, expressed as bits per symbol.

There are several trends to note in Table 7. First, as K is increased across each row, compression effectiveness generally increases, and by $K = 16$ the nominal compression limits have been reached in all cases. Second, note the advantage that the shuffled frames in the lower half of the table have compared to the corresponding contiguous ones in the upper half – while they converge to the same compression rates, the shuffled frames do so more quickly. This difference is especially pronounced when M and K are both small, that is, when *sml* is small. Third, note that the approximated frequency counts in which M is required to be a power of two can provide compression close to the ideal, especially if shuffled frames are employed. Finally, observe how in the case of the range-allocated frames, exact multiplication of the underlying frequency distribution to get $c(\cdot) = \langle 9, 9, 6, 3, 3\rangle$ does not alter the compression rate, but that in the shuffle-allocated arrangement, it does, in essence achieving the same effect as cyclicly replicating the frame via an increased value of K.

Increasing the radix also provides more precise representations. For example, the configuration shown in Table 6, with $K = 1$ and $radix = 16$, achieves compression of 2.185 bits per symbol for the same test file using the $M = 16$ approximate distribution $\hat{c}(\cdot) = \langle 5, 4, 3, 2, 2\rangle$, and achieves 2.172 bits per symbol for the $M = 32$ approximate distribution $\hat{c}(\cdot) = \langle 10, 10, 6, 3, 3\rangle$.

3.8 Prelude Representation

The dominant component of the prelude for rANS is a list of n integers that sum to m, one $c(s)$ value per symbol in $s \in \Sigma$, meaning that the prelude cost need not be any higher than it is for arithmetic coding. If scaled counts are being used, with $M > m$, the computation shown in Figure 8 can be executed by the decoder too, meaning that the original counts can be transmitted rather than the scaled ones. Finally, it might also be valid to choose $n \le M < m$, to trade prelude storage cost against message storage cost. In this case, it should be the $\hat{c}(\cdot)$ values that are placed in the prelude, rather than the $c(\cdot)$ numbers. For example, a Huffman code, which is simply an ANS code in which each $\hat{c}(s)$ is a power of two and M is also a power of two, has a reduced prelude requirement exactly because of the limited palette of $\hat{c}(s)$ values involved. For short sequences that may well result in a better overall compression rate, once the cost of the prelude is also included [4].

If a shuffled symbol ordering is being used, then the prelude must either include details of the exact permutation used in the $symbol[\cdot]$ array, or there must be an agreed protocol for distributing the symbols across the frame. For example, encoder and decoder might use the same random number generator and same seed, or might employ the same deterministic procedure for assigning the $\hat{c}(s)$ instances of s across the frame. The array $rank[\cdot]$ is then readily computed from $symbol[\cdot]$ before decoding commences. Dubé and Yokoo [14] provide a detailed analysis of techniques for constructing shuffled ANS frames.

3.9 Backwards and Forwards

The descriptions provided in Figures 2 and 4 both encode the message sequence σ in the forwards description to generate a sequence of symbols over the channel alphabet that ends with the final state. To decode, that channel sequence is consumed backwards, from last to first, to generate a reconstituted source sequence that is then reversed before being output. That is, using $reverse(\cdot)$ to indicate a sequence reversal, and $store(\cdot)$ to represent the act of storing or transmitting a compressed sequence, the decoder regenerates the input sequence σ via:

$$\sigma = reverse(ANS\_decode(reverse(store(ANS\_encode(\sigma))))) .$$

But it is also possible to have the two reversals carried out by the encoder. In particular, if the input sequence σ is reversed before being encoded, and the compressed message is reversed before it is stored, it is possible for the decoder to process its input from left to right and also write its output

from left to right, which provides a faster decoding arrangement:

$$\sigma = ANS\_decode(store(reverse(ANS\_encode(reverse(\sigma))))) .$$

Care needs to be taken in both approaches to ensure that network byte ordering ("endian-ness") issues are handled correctly if compressed messages are to be portable across architectures. Taking $radix = 2^8$ and using bytes as the input/output units is also helpful in this regard.

Asymmetric numeral systems can also be used with adaptive models such as PPM [7, 9] and DMC [11], where multiple contexts are in operation, and/or evolving probability estimates are employed. With arithmetic coding, this flexibility can be achieved on-the-fly, assuming a data structure that allows cumulative symbol frequencies to be updated after each symbol is coded. With ANS, blocks of data must be processed and the individual "coding decisions" involved in each block stacked, and then actually coded in reverse order once the end of the block has been reached. The decoder replays the same sequence of coding decisions once it has received the whole ANS block, hence following the same path of model evolution as the encoder did. Similar data structures as are used for adaptive arithmetic coding are required to maintain correct symbol counts and cumulative symbol costs, in dynamic $base[\cdot]$ and $symbol[\cdot]$ arrays. This approach relies on blocks of data being processed, and true "stream mode" adaptive compression is not possible with ANS the way it is with arithmetic coding.

4 LARGE-ALPHABET ANS CODING

Having reviewed Huffman and arithmetic coding, and provided a detailed description of ANS entropy coding, we now turn to our second objective: the use of ANS to represent sequences over very large source alphabets. In particular, we assume that the input is a sequence of m unsigned integers (`uint32_t`), rather than the more usual sequence of unsigned bytes (`uint8_t`).

4.1 Reduction to Bytes

One obvious approach is to consider the m integers to be a sequence of $4m$ bytes, and then compress them directly. Indeed, this is how a standard compression tool might approach such data, and we include two such reference implementations in the results presented in Section 5. Moffat and Petri [34] explore another option, suggesting that integer data be first processed by the VByte mechanism, which represents integers as variable-length byte sequences, and then those bytes taken as input to a subsequent byte-oriented entropy coder. Their VByte+ANS approach allows rates of less than eight bits per symbol to be achieved [34], that is, better than the lower limit that applies if VByte is used alone.

4.2 Large Frames

If $n > 256$ then neither of those two byte-reduction approaches allow the entropic bound to be reached, since both permit "muddled" byte-level probability distributions to emerge, a consequence of the way in which the individual byte values get overloaded.

On the other hand, an ANS coder can be employed directly on the integer sequence, with no modification required to the approach that was illustrated in Figure 3 and defined in Figure 4. The requirements are clear: in the encoder, a $base[\cdot]$ array of size n and a $shuffle[\cdot]$ array of size M; and in the decoder, the same $base[\cdot]$ array, and a $symbol[\cdot]$ array and a $rank[\cdot]$ array, both of size M. In other words, an ANS decoder of the type described in Section 3.4 requires $2M + n$ words of memory. (The two table-based ANS implementations described in Section 3.5 are more costly.)

A typical large-alphabet coding application might have $n = 10^6$ and $|\sigma| = m = 10^7$, and in this case, an ANS decoder may require in excess of 80 MiB of memory. While not a large amount of memory by current standards, the non-sequential nature of the array accesses required by the

```
1: function ANS_fold(s, fidelity, radix):
2: offset ← 0
3: threshold ← radix · 2^(fidelity−1)
4: while s ≥ threshold do
5:     digit ← s mod radix
6:     append digit to the output buffer
7:     s ← s div radix
8:     offset ← offset + (radix − 1) · (2^(fidelity−1))
9: return s + offset
```

Fig. 9. "Folding" a non-negative symbol number s to compute and return a unique bucket number containing a minimum of *fidelity* leading bits that uniquely differentiate that bucket, together with a succession of output units each $0 \ldots radix - 1$.

decoding function in Figure 4 mean that cache misses will play a key role in determining overall execution time, and that throughput may be affected. In this scenario, scaling the frequency counts to a new value $n \le M < m$ might be a useful way of trading compression effectiveness against both memory space and decoding throughput.

Other applications might involve $n = 10^8$ or $n = 10^9$, and messages that are sparse over the available alphabet range (that is, a majority of symbols for which $c(s) = 0$). In these cases memory space might be a problem even without considering the impact of cache misses on decoding throughput. Nor is condensing the alphabet via a remapping, so that only the symbols between 0 and n that are used in σ get allocated space in the $base[\cdot]$ array a solution, since the process of resolving symbol numbers through a bitvector-based rank/select process that converted between "internal" values and "external" ones would also risk equal volumes of cache misses.

4.3 Symbol Folding

Based on those observations, we now describe a third approach to this balancing act between memory space and decompression speed on the one hand, and compression effectiveness on the other. The underpinning idea is provided by the Elias γ code [18], in which integer $x \ge 1$ is represented as the concatenation of two parts, a binary magnitude (coded in unary), and a mantissa of that many bits, coded in binary. An obvious generalization would be to code the magnitude of each symbol as an ANS value, rather than via unary codes, with the sequence-derived frequency of occurrence of each magnitude part determining the cost of coding it. In this approach, a $base[\cdot]$ array of just 33 elements would suffice to cover all possible 32-bit unsigned integers. The binary suffixes would then be stored as themselves, bitstrings of the specified length, interspersed in the output stream amongst the corresponding ANS-code digits. Fraenkel and Klein [19] noted a similar possibility, proposing the use of Huffman codes to represent the magnitudes of the values making up a long sequence, with each Huffman-coded magnitude followed by its binary part.

The Elias γ "powers-of-two" arrangement is, however, a coarse categorization. For example, the four values between 4 and 7 are all assigned the same imputed probability in this approach; similarly, all values between 1024 and 2047 appear in the same bucket; and so on. Figure 9 describes a more fine-grained version of the same underlying idea, in which we introduce two further generalizations: first, the output radix might not be binary bits (as was already the case in Figure 4); and second, the number of leading bits that must be preserved as an accurate differentiator between buckets is specified by a second parameter denoted *fidelity*. The interaction between these two parameters means that the "mantissa", or suffix parts, of each value are represented as digits between 0 and

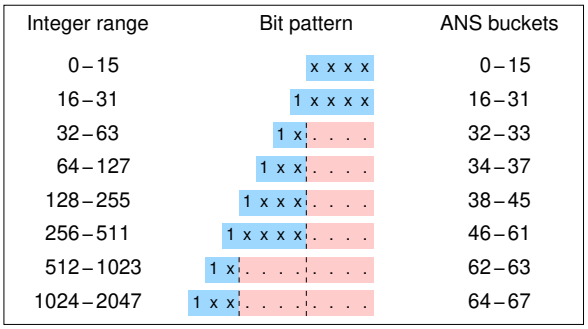

Fig. 10. Example of symbol folding, assuming that four-bit output symbols are being used (*radix* = 16), and that a minimum of two bits of leading precision are required to maintain the desired level of fidelity in the ANS probability estimates (*fidelity* = 2). Each "x" indicates a bit that contributes to the ANS bucket selector, and each "." indicates a bit that is part of one of the 4-bit output units.

radix − 1, and that the bucket sizes grow more slowly than in the Elias γ code, with a greater number of low-valued symbols being allocated to buckets of their own. In this generalized arrangement the Elias δ partitioning corresponds to *radix* = 2 and *fidelity* = 1, that is, when both parameters take their smallest values.

Figure 10 provides an example of this process, supposing that *radix* = 16, and hence that the output units are 4-bit nibbles; and that *fidelity* = 2, and hence that each bucket contains an integer range of values that all have both the same number of trailing mantissa digits, and the same first two bits when expressed in binary. In this configuration all values between 0 and 31 are assigned their own unique ANS codes, and no overloading occurs for these symbols. There are then thirty buckets each of size sixteen, indicated by the ANS folds 32–61; thirty buckets each of size 256, indicated by the ANS folds 62–91; and so on. With these parameters the value 4×10^9, near the top of the range for an unsigned 32-bit integer, is mapped to ANS fold number 224, and the ANS-coded fold identifier is followed by seven radix-16 digits (to be precise, 14, 6, 11, 2, 8, 0, 0). If a higher fidelity level is used, more symbols are allocated to unique folds, and the buckets are smaller for longer. For example, with *radix* = 16 still, but *fidelity* = 5, values 1–255 are in singleton buckets; the next 240 buckets each contain 16 values and cover the input integers 256–4095; and the value 4×10^9 maps to ANS fold number 1678 and is followed by six radix-16 digits (6, 11, 2, 8, 0, 0). If a higher *radix* is used there are again more singleton buckets: with *radix* = 256 and *fidelity* = 5, values up to 4095 are uniquely coded; and the value $s = 4 \times 10^9$ corresponds to ANS fold 12,478 and is followed by three bytes (107, 40, 0). These numbers are all exact, and computed using the process described in Figure 9; in approximate terms for large values of s the ANS fold can be estimated via

$$\mathit{ANS\_fold}(s, \mathit{fidelity}, \mathit{radix}) \approx \left\lceil \log_{\mathit{radix}} s + (\mathit{fidelity} - 1)/(\log_2 \mathit{radix}) \right\rceil \cdot (\mathit{radix} - 1) \cdot (2^{\mathit{fidelity}-1}),$$

which helps explain the interactions between these three quantities.

Instances of this general-purpose mapping have been used in the past. In particular, the relationship with the Elias γ code [18] was noted above, as was the Huffman-based proposal of Fraenkel and Klein [19]; both of these can be regarded as being examples in which *radix* = 2 and *fidelity* = 1. Symbol folding has also been employed in implementations of compression systems. For example, the well-known tool GZip makes use of the DEFLATE mappings[1] to encode backwards copy offsets, an arrangement that corresponds to the use of *fidelity* = *radix* = 2. In GZip a semi-static Huffman

[1]https://en.m.wikipedia.org/wiki/DEFLATE, accessed 2 February 2020.

code is used to represent the fold numbers, and the trailing digits are always sequences of bits. The generalization that we have introduced here allows the use of other combinations, including desirable options in which *radix* = 256, and the trailing digits are sequences of bytes.

Note also that the folding process has the potential to substantially reduce the size of the prelude, an approximation process that was already alluded to in Section 3.8.

4.4 Partial Alphabet Re-Ordering

The folding process assumes that the most frequent symbols in the alphabet – the ones for which it is most important to have accurate probability estimates – have low symbol numbers. For many applications that will be the case. But it is also possible for the most frequent symbols to be elsewhere in the alphabet (Section 5.2 describes one such scenario), and when that occurs, the enforced "sharing" that occurs within buckets might still lead to degraded effectiveness. To further mitigate against that, we also employ a technique proposed by Culpepper and Moffat [12]. The idea is to identify a subset of high-frequency symbols that are explicitly mapped to early symbol numbers, with all other symbol numbers displaced by the size of that set. For example, if two symbols, say 1526 and 7543, are both frequent, they might be mapped to symbols 1 and 2 respectively, rather than left in shared buckets, and all other symbols s coded as adjusted values $s' = s + 2$. Two "holes" are created, at 1528 and 7545, symbols s' that will never occur, but overall the gains arising as a result of maintaining clean probabilities for high-frequency symbols are likely to completely outweigh the small cost of the holes.

The set of selected symbols must be identified in the prelude, and a small additional table maintained in the decoder that contains the set of re-ordered symbols. But both are much less costly than maintaining the full permutation vector that would be required by a complete re-mapping of the alphabet into decreasing probability order. In the experiments described in Section 5, the top k most frequent symbols are identified, and mapped into the first k positions, where k is the number of singleton buckets created by the ANS folding process with parameters *radix* and *fidelity*.

4.5 Implementation Details

The ANS encoding and decoding operations can be expressed succinctly in pseudocode. However, achieving high throughput levels also requires attention to a range of engineering details.

Output, Normalization, and Divisions. Decoding speed relies on efficient output, rapid state normalization, and avoidance of integer divisions. The implementations measured in Section 5 employ a 64-bit *state* variable and *radix* = 2^{32} to minimize the number of input/output operations. At the same time, the frame size (M) is restricted to powers of two so that all mod and div operations (see Figure 4) can be replaced by shifts and masks. The final *state* value is written uncompressed as a 64-bit integer, despite the fact that it can be expected to be slightly biased towards smaller values; and no attempt is made to exploit the (minimum of) 32-bits of capacity available in the initial value of *state*.

Some of the tested implementations also include the ANS folding technique. When this facility is enabled, the *radix* using in the folding process is 256 = 2^8. Note that the 8-bit output values generated by the ANS folding process can be interleaved with the 32-bit symbols generated by the encoding renormalization loop without any additional overhead being incurred.

Encoding and Decoding Direction. As discussed in Section 3.9, ANS-based coders decode symbols in last in, first out order. To prioritize decoding speed, our implementations encode symbols in reverse order and decode in correct input order.

Pre-Computation. Several of the required values can be pre-computed and stored, offering trade-offs between reduced computation time and additional memory. For example, the symbol upper bound $K \cdot radix \cdot c(\sigma_i)$ required at step 7 of Figure 4 can be retained in an array of 64-bit values, indexed by symbol number σ_i. Similarly, if ANS folding is taking place, the inverse of *ANS_fold* can be pre-computed, and recovered in a branch-free manner by masking/shifting the required number of input symbols from the next 32-bit word in the decoder's input stream.

Prelude Representation. When n is large and comparable in magnitude to m, the prelude associated with each block of compressed data might be significant fraction of the compressed block. The prelude can be thought of as consisting of two components [43]: a subalphabet selection vector, which indicates which symbols in $0 \ldots n-1$ that occur in this block; followed by a list of their frequencies $c(\cdot)$ (or scaled frequencies $\hat{c}(\cdot)$ if $M < m$). The ANS implementations measured in Section 5 represent the n-sequence $\langle 1 + c(s) \rangle$ using binary interpolative coding [35, 41], thereby handling both components in an economical way that exploits the likely pattern of low symbol identifiers being more likely to occur that large ones. An alternative is to transmit the n-sequence $\langle c(s) \rangle$ using an entropy coder. This is easily done via a recursive call to the ANS encoding function, with a base case to handle the (eventual) situation $n' \ll m'$, where $m' = n$ is the length of the prelude sequence, and n' is the maximum symbol frequency that has arisen, $n' = \max_{0 \le s < n}\{c(s)\}$.

Minimizing Frame Size. Decoding a symbol (Figure 4) requires a lookup in each of the $rank[\cdot]$ and $symbol[\cdot]$ arrays, as well as identification of $c(s)$. To minimize the space cost, and to at the same time minimize the number of cache misses that might occur, these three components are combined into a single array D of size M. Each element of D is a structure containing three components: a 32-bit value $D[r].symbol$, equal to $symbol[r]$; a 16-bit value $D[r].count$, equal to $c(symbol[r])$; and a 16-bit value $D[r].rank$, equal to $rank[r]$. When $n < m$ this arrangement involves a level of redundancy, since each *count* field is stored that many times. But retaining the three decoding elements as a single 64-bit structure in one array means that at most one cache miss can be incurred per decoded symbol.

The decision to employ a combined 64-bit structure for the three arrays, and the use of shift/mask operations to accomplish the div and mod computations, have implications for the scaling regime. In the implementation we carry out scaling (Figure 8) subject to three constraints: (a) the maximum scaled frequency $\hat{c}(s)$ (and hence all of the $rank[\cdot]$ values) must be less than 2^{16}; (b) the frame size M must be a power of two; and (c) the cross-entropy compression loss (Equation 2) caused by the choice of M should ideally be less than some multiplicative tolerance θ.

Hence, to determine M and $\hat{c}(s)$, the following process is carried out: starting with $M = 2^{\lceil \log_2 n \rceil}$, cross-entropy costs are computed. If the relative cross-entropy overhead exceeds θ bits per integer, M is doubled, and the scaling process applied again. This doubling process stops either when the largest value $\hat{c}(s)$ exceeds 2^{16}, or when the cross-entropy overhead is less than θ, or when M reaches 2^{27}. The final of these three constraints means that at most $2^{27} \times 64$ bits = 1 GiB of memory can be consumed by the array $D[\cdot]$ that is employed during decoding.

Interleaved Compression. ANS coding has also received significant attention in the form of blog posts and carefully tuned systems developed by skilled implementors, including Fabian Giesen[2], Charles Bloom[3], and Yann Collet[4]. One particularly insightful technique that has emerged from their work is that it is possible to interleave the output of multiple ANS compressors [21]. This allows

[2]For example, https://fgiesen.wordpress.com/2014/02/02/rans-notes/, accessed September 2019.
[3]For example, http://cbloomrants.blogspot.com/2014/02/02-01-14-understanding-ans-3.html, accessed September 2019.
[4]For example, http://fastcompression.blogspot.com/2014/01/fse-decoding-how-it-works.html; accessed September 2019, and https://github.com/Cyan4973/FiniteStateEntropy, accessed September 2019.

multiple ANS state computations to be concurrently active, each processing different symbols of the input stream. A form of loop unrolling, CPU instruction pipelining can then substantially enhance compression and decompression throughput. In the large-alphabet implementations explored in Section 5, four ANS computation pipelines are employed, all referencing the same frame description, with each of the four processing every fourth input symbol, and their 32-bit output units interleaved as they are emitted. At the same time, the bytes associated with the *ANS_fold* computation (with *radix* = 256) are directly inserted into the output stream as they are generated. During decoding the corresponding number of bytes is read from the compressed stream, allowing recovery of the original symbol.

5 EXPERIMENTS

The large-alphabet entropy coders discussed above are evaluated by comparing to several other compression implementations over a variety of synthetic and real-world inspired datasets.

5.1 Baselines and Methodology

Baselines. We compare to a variety of baselines that range across purpose-built large-alphabet entropy coders; modified byte-based coders; and specialized codecs for large integer sequences:

- huff0, a highly engineered SIMD-enabled semi-static Huffman coder over byte-based input sequences, that is, with $n = 256$, written by Yann Collet and available at https://github.com/Cyan4973/FiniteStateEntropy;
- FSE, a highly engineered table-based semi-static ANS coder over input sequences treated as bytes, also by Yann Collet, and available as part of the same FiniteStateEntropy package;
- shuff, a semi-static large-alphabet Huffman coder over input sequences of 32-bit words, written by Andrew Turpin and available at https://github.com/turpinandrew/shuff;
- arith64, a semi-static large-alphabet arithmetic coder over input sequences of 32-bit words, that uses 64-bit arithmetic and byte-based renormalization [40];
- VByte, an implementation for compressing 32-bit integers [45] taken from the FastPFor library of Daniel Lemire available at https://github.com/lemire/FastPFor; and
- OptPFor, another 32-bit integer compression codec [47, 49], with implementation from the FastPFor library.

Two "combination" methods were also included, denoted VByte+huff0 and VByte+FSE; these use VByte to transform the sequence of 32-bit input integers into a byte sequence which is can then be provided as input to the corresponding byte-based entropy coder [34].

Methods. We provide new ANS implementations[5] which implement the large-alphabet techniques discussed in Section 4:

- ANS, an rANS implementation that makes use of a frame of size M, with each symbol allocated the space in it according to its relative frequency, and including the various techniques described in Section 4.5;
- ANSfold-f, which adds the *ANS_fold* mapping with fidelity parameter f (and *radix* parameter 256) to reduce the frame size M, but at the risk of symbol overloading and hence mis-estimation of probabilities (Section 4.3); and
- ANSfold-f-r, which further adds partial reordering of the alphabet so that the k ANS fold buckets that contain only a single value all correspond to one of the k most frequent symbols, regardless of where those symbols appear in the alphabet (Section 4.4).

[5]See https://github.com/mpetri/ans-large-alphabet.

Name	m	n'	n	$\mathcal{H}(\sigma)/m$
uni-12	100,000,000	4096	4096	12.00
geo-0.4	100,000,000	38	39	2.43
geo-0.9	100,000,000	9	9	0.52
zipf-20	100,000,000	1,048,413	1,048,576	13.44
newsdocs-w	100,000,000	386,987	386,987	10.97
bwtmtf-w	100,000,000	352,003	386,987	8.48
rlz-d64off	100,000,000	6,165,676	67,109,799	21.19
rlz-d64len	100,000,000	1062	1336	4.97

Table 8. Dataset statistics. The additional quantity n' is the number of distinct symbols in the range $0 \ldots n-1$ that actually appear in the sequence.

In all three ANS implementations four states are interleaved, and the cross-entropy scaling ratio, which is part of the process for determining the frame size M, is set at $\theta = 1.001$.

Experimental Context and Metrics. All experiments were performed on a Intel Xeon 6144 CPU equipped with 512 GiB RAM and 32 MiB cache. While by no means a commodity processor, at time of writing this hardware is three years old, and comparable performance and cache capacities are available for around $400[6]. The C++ code was compiled using gcc 7.3 with O3 optimizations enabled. Each compression scheme receives the input as a vector of m `uint32_t` integers and constructs a vector of `uint8_t` bytes.

In-memory encoding and decoding times are measured, recording the *minimum* execution time across five runs, with encoding and decoding rates reported in integers per second, and compression effectiveness (including all costs associated with prelude storage, but not including the size of the program binary) reported in units of bits per integer. To assess consistency, all of the efficiency experiments were performed five times. There were no cases in which the measured deviations relative to the reported values were greater than 1.1%, across all of the methods and all of the datasets. Moreover, the relative ordering between methods was never affected by the small variations in timings that did arise. Hence we have confidence in the generality of the throughput results presented in the tables in Section 5.4. The experiments are self-contained and can be reproduced using the test harness that is included with the implementations.

Many compression systems make use of SIMD instructions to accelerate throughput – see, for example, the work of Lemire and Boytsov [30]. However it is difficult to properly compare such highly-engineered implementations against other systems without also investing similar tuning and development effort into the set of comparator systems. To avoid possible unfairness, we primarily compare against state-of-the-art non-SIMD implementations, knowing that they might be further accelerated by the application of careful hardware-specific tuning.

5.2 Datasets

To cover the wide range of scenarios in which an entropy coder might be employed, we employ both synthetic and "natural" sequences. A summary of the dataset characteristics is shown in Table 8.

Artificial Data Sequences. We employ several artificial, randomly generated sequences based on three typical probability distributions:

[6]https://cpu-benchmarks.com/cpu/intel-xeon-gold-6144-3-50ghz/, accessed 2 February 2020.

Method	uni-12	geo-0.9	rlz-d64len	bwtmtf-w	zipf-20	newsdocs-w	rlz-d64off
huff0	17.29	4.11	9.16	11.20	16.70	15.84	28.95
FSE	17.25	0.74	8.54	10.55	16.62	15.76	28.94
VByte	15.75	8.00	8.21	10.99	15.30	14.34	31.64
VByte+huff0	14.00	1.11	5.15	8.74	13.88	13.05	29.35
VByte+FSE	13.94	0.52	5.11	8.76	13.85	13.03	29.23
OptPFor	12.25	1.11	6.02	10.39	16.61	14.31	28.72
shuff	12.00	1.11	5.00	8.52	13.52	11.02	21.70
arith64	12.00	0.52	4.98	8.50	13.50	10.99	21.80
ANS	12.00	0.52	4.98	8.50	13.48	10.99	21.71
ANSfold-1	12.00	0.52	4.98	8.49	13.45	11.89	25.99
$\mathcal{H}(\sigma)/m$	12.00	0.52	4.97	8.48	13.44	10.97	21.19

Table 9. Compression effectiveness in bits per integer for baseline and large-alphabet ANS methods. The last row shows the entropy-based bound, without any allowance for storage of the prelude.

uni-d, integer values uniformly sampled from the interval $[0, 2^d - 1]$;
geo-ϕ, integer values sampled from a geometric distribution, with $P(x \mid \phi) = \phi \cdot (1 - \phi)^x$; and
zipf-d, integer values in $[0, 2^d - 1]$ sampled with probabilities given by $P(x \mid d) = 1/(S_d \cdot (x + 1))$, where $S_d = \sum_{i=1}^{2^d} (1/i)$.

Natural Data Sequences. Real-world sequences occurring in standard large-alphabet compression scenarios are also included in the test suite:

newsdocs-w, a standard English text corpus consisting of text extracted from online newspapers in 2016, and freely available at http://data.statmt.org/news-crawl/en-doc/, parsed into a sequence of word numbers where each word is assigned a unique integer in order of first appearance;
bwtmtf-w, the result of applying the Burrows-Wheeler Transform [8] to the word-parsed sequence arising from the newsdocs collection, and then applying a Move-To-Front transformation;
rlz-d64len, the "copy lengths" arising when the Relative Lempel Ziv compression scheme [23, 24, 38] using a 64 MiB uniformly-selected dictionary is applied to the first 10 WARC files of the freely available 2018 NEWS CommonCrawl (53 GiB in total); and
rlz-d64off, the sequence of Relative Lempel Ziv "dictionary offsets" corresponding to the file rlz-d64len.

Each of these four files was truncated to the indicated length of m = 100,000,000, with further values discarded.

5.3 Compression Effectiveness

Table 9 shows a subset of the full suite of compression effectiveness outcomes. All of the integer based entropy coders (shuff, arith64, ANS and ANSfold) represent the uni-12 dataset without compression loss relative to the entropy limit. On the other hand, the entropy coders that compress the `uint32_t` input stream as a stream of bytes (huff0 and FSE) incur a substantial effectiveness penalty, highlighting the risks that arise if there is a mismatch between the data and the modeling assumptions. The penalty can be partially reduced by preprocessing the integer inputs with the VByte encoder before performing entropy coding on the resulting byte stream (VByte+huff0 and

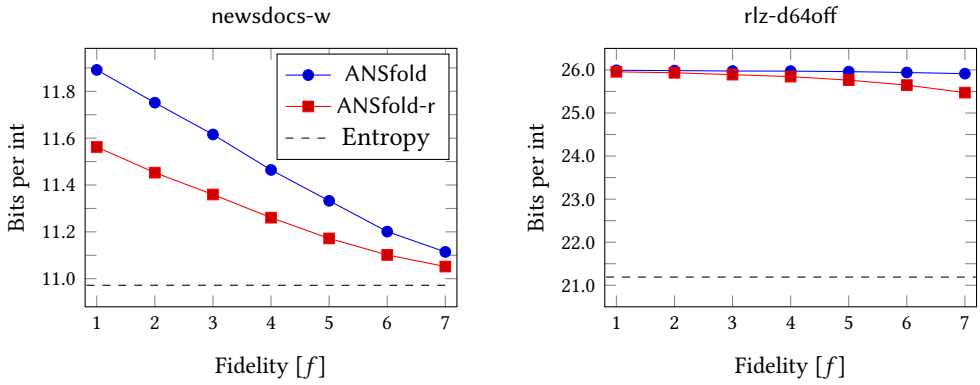

Fig. 11. Compression effectiveness for the ANSfold and ANSfold-r methods for different fidelity thresholds $f = 1$–7 for files newsdocs-w (left) and rlz-d64off (right).

VByte+FSE). As expected, the highly engineered locally adaptive integer sequence coder OptPFor also achieves compression rates close to the entropy limit for uniform data.

The advantage of ANS-based entropy coders (VByte+FSE, ANS, ANSfold) and the arithmetic coder (arith64) become clear on the geo-0.9 test file, with compression that equals the entropy rate of 0.52 bits per integer. All other methods require at least one bit per integer.

Continuing across the table, the two very large-alphabet files bwtmtf-w and zipf-20 with strictly decreasing probability distributions can be compressed with minimal compression loss by all of the integer entropy coders, with the folding component of the ANSfold method resulting in a smaller frame size than for the ANS and arith64 approaches, and hence a slightly reduced prelude overhead. The two VByte-based methods also compress these collections well, with ≈ 0.4 bits per integer loss.

For the two remaining large alphabet test cases (newsdocs-w and rlz-d64off), ANSfold does not perform as well as the other integer based entropy coders, because the probability distributions are not decreasing. This issue is explored in Figure 11. Recall that the fidelity parameter determines the number of leading bits that are specific to each of the ANS buckets, and that large values of f imply a greater number of singleton buckets. Because of the way it was constructed, the probability distribution for newsdocs-w has a tendency for small integers to be common across the rest of the file, but that is by no means assured, and small integers might also be assigned to infrequent words. Thus, having a larger ANS frame that models more values exactly leads to improvements in compression effectiveness – the effect shown by the blue line in the left pane of Figure 11. In addition, explicitly remapping the most frequent symbols to small integers (ANSfold-r, the red line) yields faster convergence towards the entropy limit, albeit at the slight added cost of needing to store the remapped integers in the prelude.

For the rlz-d64off dataset, shown in the right pane of Figure 11, increasing f does not have the same effect, and the gap to the entropy limit remains large. Even remapping the frequent symbols only achieves a modest improvement. This behavior arises because the RLZ offset values have no correlation with their probability – the high-frequency symbols can appear equally over the complete 64 MiB dictionary range. Moreover, inside each bucket, both ANSfold and ANSfold-r assume a uniform distribution, but there are offsets that are common and (in part, as a result) other offsets nearby that don't occur at all. In other words, the sparseness of the data and the lack of detailed frequency information right down to the integer level, and right across the full range of the alphabet, erodes the effectiveness of the two ANSfold methods on this test file.

Method	geo-0.4		rlz-d64len		zipf-20		newsdocs-w	
	enc.	dec.	enc.	dec.	enc.	dec.	enc.	dec.
VByte	1288	1275	1124	1113	206	212	301	300
VByte+huff0	511	668	458	557	138	144	178	185
VByte+FSE	360	444	338	414	115	132	143	167
OptPFor	30	972	17	902	4	411	5	424
shuff	297	255	249	220	89	130	168	177
arith64	154	47	96	29	61	10	91	14
ANS	127	557	118	374	60	39	90	36
ANSfold-1	123	475	112	393	60	329	69	317
ANSfold-5	120	477	113	330	65	298	74	217
ANSfold-1-r	101	479	93	398	35	337	50	343
ANSfold-5-r	100	480	93	331	38	295	61	303

Table 10. Encoding and decoding throughput rates, in millions of integers per second, for a range of datasets.

5.4 Encoding and Decoding Throughput

Table 10 reports encoding and decoding throughout rates for four of the test files: geo-0.4, a small, highly skewed dataset; rzl-d64len, a medium-scale dataset; and zipf-20 and newsdocs-w, two large-alphabet datasets. The two byte-wise entropy coders (huff0 and FSE) are not included in Table 10, because in processing bytes rather than integers, they handle four times as many symbols, and comparison would be unfair. Note also that the new ANS and ANSfold implementations were assembled with an emphasis on decoding speed, and several optimizations to improve encoding speed (including replacing divisions by shifts) are possible that would likely improve their encoding performance relative to the values shown in the table.

For geo-0.4 the ANS implementations achieve approximately twice the decoding throughput of the shuff software, in part a consequence of the small frame size involved, fitting completely in cache; and in part because of the interleaving. The two hybrids using VByte preprocessing also operate quickly. The fastest decoding is provided by VByte and OptPFor, but with the latter providing the slowest encoding speed. The arith64 implementation achieves credible encoding speeds, with all multiplicative operations carried out via shift/mask replacements; but decodes slowly, because of the 64-bit full division that is required for each symbol.

The medium alphabet dataset (rzl-d64len) shows similar characteristics as geo-0.4, but with ANS decoding speeds decreasing because of the larger frames. Similar tendencies can be observed for shuff, while VByte and OptPFor are relatively unaffected by the increase in alphabet size.

Looking at the two large alphabet test files, zipf-20 and newsdocs-w, the ANS decoding throughput becomes markedly slower, because of the large amount of memory required for a full ANS frame, and the cache misses that occur. On the other hand, the ANSfold implementations still achieve good decoding throughput, because their memory footprints remain small. The arith64 mechanism also slows as the alphabet becomes larger, because its implementation employs a binary search in an array of cumulative frequencies, rather than a table lookup the way the full-frame ANS does. Note that the fidelity parameter f affects the encoding and decoding speed of ANSfold, with higher values of f leading to bigger data structures and slower decoding.

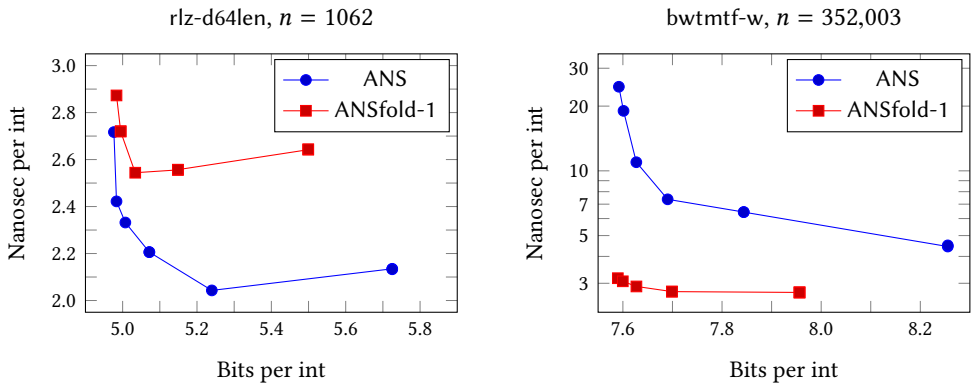

Fig. 12. Decoding speed for ANS and ANSfold-1 for a range of frame sizes, determined by varying the cross-entropy tolerance θ, for files rlz-d64len (moderate n) and bwtmtf-w (large n).

5.5 ANS Modeling Accuracy

Table 9 suggests that the size of the underlying ANS frame has a non-trivial impact on ANS decoding throughput. The test implementation chooses the frame size by iteratively increasing the frame size M such that the approximated probabilities $\hat{c}(s)/M$ result in relative compression inefficiency due to modeling inaccuracy at most a constant factor θ from $\mathcal{H}(\sigma)$. Allowing for more modeling inaccuracy by increasing θ results in smaller ANS frames being constructed, which can improve throughput. This tradeoff is explored in Figure 12, where decoding speed and compression effectiveness are plotted against each other as θ is varied across $\{1.001, 1.005, 1.01, 1.02, 1.04, 1.08, 1.16, 1.32\}$, with speed shown in nanoseconds per integer.

For the rlz-d64len dataset (in the left pane), with a relatively small alphabet, when the threshold is small (1.001 to 1.01), the frame gets enlarged so that it accurately models the true probabilities, and compression loss is minimal. Thereafter, as the threshold increases, decoding times decrease because of the smaller frames employed, while effectiveness is eroded. The ANS implementation utilizes frames of size $M = 65{,}536$ for threshold 1.001, decreasing to $M = 2048$ for $\theta = 1.32$. Method ANSfold-1 employs $M = 8192$ to $M = 512$ across the same spectrum of θ. Compared to the ANS reference point, ANSfold-1 incurs a speed penalty, because the folded alphabet is already small, and there is slightly more processing required per decoded symbol. This behavior is reversed for the large alphabet dataset bwtmtf-w (the right frame). The ANS approach requires large frames ($M = 8{,}388{,}608$ for $\theta = 1.001$, decreasing to $M = 262{,}144$ for $\theta = 1.32$) even when θ is relatively large, whereas ANSfold-1 still employs frames that are compact ($M = 8192$ for $\theta = 1.001$, and $M = 1024$ for $\theta = 1.32$) and support fast decoding when θ is small, with per-integer decoding times unchanged compared to more moderate alphabet sizes.

5.6 Pseudo-Adaptive Coding

Unless it is certain that a large input file is homogeneous, it can be beneficial to partition long input sequences into blocks, and process each block independently, so as to exploit any available local variations in symbol frequencies. Splitting the input into blocks also reduces the buffer space required in the lead-up to and during the required *reverse*() operations. Partitioning big files into smaller components usually reduces the size of the subalphabet required in each block, but at the same time creates tension between total prelude cost (which must be summed over the blocks) and decreased code cost (because each block has a smaller alphabet and more precise probabilities).

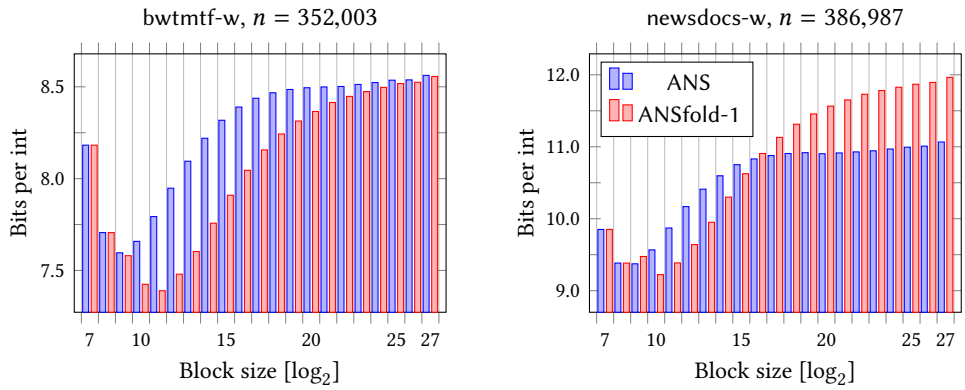

Fig. 13. Compression effectiveness for ANS and ANS-fold for block sizes ranging from 2^7 to 2^{27}.

This tradeoff is explored in Figure 13, which shows block-by-block processing via ANS and ANSfold-1 of two large-alphabet files (bwtmtf-w and newsdocs-w), with block-sizes of 2^b for $b \in [7, 28]$. As expected, both methods approach the compression rates reported in Table 9 when the block-size is large. But there are marked differences when smaller block-sizes are employed. In both cases, ANSfold-1 outperforms ANS, a consequence of the former's smaller per-block prelude costs. Figure 11 already explored how increasing the fidelity f of ANSfold helped match the compression effectiveness of ANS for the newsdocs-w dataset; what is clear from Figure 13 is that block-by-block processing is a second alternative that is just as potent, providing not only the option of equaling the entropy (10.97 bits per integer) when a block-size of 65,536 is used, but also the option of undercutting the entropy (for both of these two non-homogeneous files, by up to 15%) when smaller block-sizes are employed. Not shown in Figure 13 is that small block sizes also suffer a throughput penalty, because of the repeated need to process the prelude and reconstruct the ANS decoding structures, making this a three-way tradeoff between block-size, effectiveness, and efficiency. In this regard, ANSfold is likely to be the preferred choice, as it provides better effectiveness across a wide range of block-sizes and also allows much faster decoding (Table 10).

6 CONCLUSIONS

Our goals with this paper were threefold: to give an accessible and cohesive presentation of the recent ANS mechanism; to describe modifications that would allow ANS to be employed for large-alphabet semi-static compression; and to measure the relative performance of these ANS variants and compare them to Huffman and arithmetic coding. Sections 3 and 4 respond to the first two of those goals, and, as a clear conclusion of this project, the results presented in Section 5 confirm that ANS entropy coding is an important technique that provides clear effectiveness and efficiency options compared to previous approaches. In particular, when coupled with the ANS-fold and ANS-fold-r mechanisms, ANS provides excellent compression effectiveness on many types of large-alphabet input data, in most cases as good as can be achieved by arithmetic coding; and maintains high decoding rates that are superior to what can be attained by canonical Huffman coding.

Software. The code that was developed during this work, and the experimental setup used to carry out the experiments, are available at https://github.com/mpetri/ans-large-alphabet.

Acknowledgment. This work was partially supported under the Australian Research Council's Discovery Projects funding scheme (grant DP200103136) and partially supported by funding from Huawei Inc. (grant HO2019041002004P103). We thank the referees for their helpful suggestions.

REFERENCES

[1] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne. 2019. Brotli: A general-purpose data compressor. *ACM Trans. on Information Systems* 37, 1 (2019), 4:1–4:30.
[2] T. C. Bell, J. G. Cleary, and I. H. Witten. 1990. *Text Compression.* Prentice Hall, Englewood Cliffs, NJ.
[3] T. C. Bell, I. H. Witten, and J. G. Cleary. 1989. Modeling for text compression. *Comput. Surveys* 21, 4 (1989), 557–591.
[4] A. Bookstein and S. T. Klein. 1993. Is Huffman coding dead? *Computing* 50, 4 (1993), 279–296.
[5] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. 2008. Reorganizing compressed text. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR).* 139–146.
[6] N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller. 2003. (S, C)-Dense coding: An optimized compression code for natural language text databases. In *Proc. Symp. on String Processing and Information Retrieval (SPIRE).* 122–136.
[7] S. Bunton. 1997. Semantically motivated improvements for PPM variants. *Computer Journal* 40, 2/3 (1997), 76–93.
[8] M. Burrows and D. J. Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report 124. Digital Equipment Corporation, Palo Alto, California.
[9] J. G. Cleary and I. H. Witten. 1984. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communications* 32, 4 (1984), 396–402.
[10] J. B. Connell. 1973. A Huffman-Shannon-Fano code. *Proc. IEEE* 61, 7 (1973), 1046–1047.
[11] G. V. Cormack and R. N. Horspool. 1987. Data compression using dynamic Markov modeling. *Computer Journal* 30, 6 (1987), 541–550.
[12] J. S. Culpepper and A. Moffat. 2005. Enhanced byte codes with restricted prefix properties. In *Proc. Symp. on String Processing and Information Retrieval (SPIRE).* 1–12.
[13] E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. 2000. Fast and flexible word searching on compressed text. *ACM Trans. on Information Systems* 18, 2 (2000), 113–139.
[14] D. Dubé and H. Yokoo. 2019. Fast construction of almost optimal symbol distributions for asymmetric numeral systems. In *Proc. Int. Symp. on Information Theory.* 1682–1686.
[15] J. Duda. 2009. Asymmetric numeral systems. *CoRR* abs/0902.0271 (2009).
[16] J. Duda. 2013. Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *CoRR* abs/1311.2540 (2013).
[17] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. 2015. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *Proc. Picture Coding Symp.* 65–69.
[18] P. Elias. 1975. Universal codeword sets and representations of the integers. *IEEE Trans. on Information Theory* IT-21, 2 (1975), 194–203.
[19] A. S. Fraenkel and S. T. Klein. 1985. Novel compression of sparse bit-strings: Preliminary report. In *Combinatorial Algorithms on Words, Volume 12 (NATO ASI Series F).* Springer-Verlag, 169–183.
[20] T. Gagie, G. Navarro, Y. Nekrich, and A. Ordóñez Pereira. 2015. Efficient and compact representations of prefix codes. *IEEE Trans. on Information Theory* 61, 9 (2015), 4999–5011.
[21] F. Giesen. 2014. Interleaved entropy coders. *CoRR* abs/1402.3392 (2014).
[22] S. W. Golomb. 1966. Run-length encodings. *IEEE Trans. on Information Theory* IT–12, 3 (1966), 399–401.
[23] C. Hoobin, S. J. Puglisi, and J. Zobel. 2011. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB* 5, 3 (2011), 265–273.
[24] C. Hoobin, S. J. Puglisi, and J. Zobel. 2011. Sample selection for dictionary-based corpus compression. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR).* 1137–1138.
[25] P. G. Howard and J. S. Vitter. 1992. Analysis of Arithmetic Coding for Data Compression. *Information Processing & Management* 28, 6 (1992), 749–763.
[26] P. G. Howard and J. S. Vitter. 1992. New methods for lossless image compression using arithmetic coding. *Information Processing & Management* 28, 6 (1992), 765–779.
[27] P. G. Howard and J. S. Vitter. 1994. Arithmetic coding for data compression. *Proc. IEEE* 82, 6 (1994), 857–865.
[28] D. A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.* 40, 9 (1952), 1098–1101.
[29] D. A. Lelewer and D. S. Hirschberg. 1987. Data compression. *Comput. Surveys* 19, 3 (1987), 261–296.
[30] D. Lemire and L. Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software Practice & Experience* 45, 1 (2015), 1–29.

[31] M. Martinez, M. Haurilet, R. Stiefelhagen, and J. Serra-Sagristà. 2017. Marlin: A high throughput variable-to-fixed codec using plurally parsable dictionaries. In *Proc. IEEE Data Compression Conf. (DCC)*. 161–170.
[32] A. Moffat. 2019. Huffman Coding. *Comput. Surveys* 52, 4 (2019), 85.1–85.35.
[33] A. Moffat, R. Neal, and I. H. Witten. 1998. Arithmetic coding revisited. *ACM Trans. on Information Systems* 16, 3 (1998), 256–294. Source code available from http://people.eng.unimelb.edu.au/ammoffat/arith_coder.
[34] A. Moffat and M. Petri. 2017. ANS-based index compression. In *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 677–686.
[35] A. Moffat and L. Stuiver. 2000. Binary interpolative coding for effective index compression. *Information Retrieval* 3, 1 (2000), 25–47.
[36] A. Moffat and A. Turpin. 2002. *Compression and Coding Algorithms*. Kluwer, Boston.
[37] S. M. Najmabadi, T.-H. Tran, S. Eissa, H. S. Tungal, and S. Simon. 2019. An architecture for Asymmetric Numeral Systems entropy decoder: A comparison with a canonical Huffman decoder. *J. Signal Processing Systems* 91, 7 (2019), 805–817.
[38] M. Petri, A. Moffat, P. C. Nagesh, and A. Wirth. 2015. Access time tradeoffs in archive compression. In *Proc. Asia Information Retrieval Societies Conf. (AIRS)*. 15–28.
[39] G. E. Pibiri, M. Petri, and A. Moffat. 2019. Fast dictionary-based compression for inverted indexes. In *Proc. Conf. on Web Search and Data Mining (WSDM)*. 6–14.
[40] M. Schindler. 1998. A fast renormalisation for arithmetic coding. In *Proc. IEEE Data Compression Conf. (DCC)*. 572.
[41] J. Teuhola. 2009. Tournament coding of integer sequences. *Computer Journal* 52, 3 (2009), 368–377.
[42] A. Trotman. 2003. Compressing inverted files. *Information Retrieval* 6, 1 (2003), 5–19.
[43] A. Turpin and A. Moffat. 2000. Housekeeping for prefix coding. *IEEE Trans. on Communications* 48, 4 (2000), 622–628. Source code available from http://people.eng.unimelb.edu.au/ammoffat/mr_coder/.
[44] J. van Leeuwen. 1976. On the construction of Huffman trees. In *Int. Colloq. Automata Languages and Programming (ICALP)*. Edinburgh University Press, Edinburgh University, Scotland, 382–410.
[45] H. E. Williams and J. Zobel. 1999. Compressing integers for fast file access. *Computer Journal* 42, 3 (1999), 193–201.
[46] I. H. Witten, R. M. Neal, and J. G. Cleary. 1987. Arithmetic coding for data compression. *Commun. ACM* 30, 6 (1987), 520–541.
[47] H. Yan, S. Ding, and T. Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proc. Conf. on the World Wide Web (WWW)*. 401–410.
[48] J. Zobel and A. Moffat. 1995. Adding compression to a full-text retrieval system. *Software Practice & Experience* 25, 8 (1995), 891–903.
[49] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. 2006. Super-scalar RAM-CPU cache compression. In *Proc. Int. Conf. on Data Engineering (ICDE)*. 59.

Received September 2019; revised February 2020; accepted April 2020