Redundancy and Optimization of tANS Entropy Encoders

Ian Blanes, *Senior Member, IEEE,* Miguel Hernández-Cabronero,
Joan Serra-Sagristà, *Senior Member, IEEE*, and Michael W. Marcellin, *Fellow, IEEE*

***Abstract*—Nowadays entropy encoders are part of almost all data compression methods, with the Asymmetrical Numeral Systems (ANS) family of entropy encoders having recently risen in popularity. Entropy encoders based on the *tabled* variant of ANS are known to provide varying performances depending on their internal design. In this paper, we present a method that calculates encoder redundancies in almost linear time, which translates in practice to thousand-fold speedups in redundancy calculations for small automatons, and allows redundancy calculations for automatons with tens of millions of states that would be otherwise prohibitive. We also address the problem of improving tabled ANS encoder designs, by employing the aforementioned redundancy calculation method in conjunction with a stochastic hill climbing strategy. The proposed approach consistently outperforms state-of-the-art methods in tabled ANS encoder design. For automatons of twice the alphabet size, experimental results show redundancy reductions around 10% over the default initialization method and over 30% for random initialization.**

***Index Terms*—Tabled Asymmetrical Numeral Systems, Entropy encoder redundancy, Optimization.**

I. Introduction

OF the diverse set of tools that are employed in data compression, entropy encoders are one of the most used tools, if not the most. They are the last stage in almost all data compression methods, after a varying set of predictors and transforms. One particular family of entropy encoders that has gained popularity in recent years is the one based on the Asymmetrical Numeral Systems (ANS) [1], [2], [3]. Some ANS encoders and decoders have been shown to allow for very fast software implementations in modern CPUs [4], [5]. This has lead to its incorporation in recent data compression standards and its use in many different cases [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. In addition, ANS-based encoders could be applicable to a very wide range of multimedia scenarios, such as an alternative to the Rice-Golomb codes employed in the energy-efficient scheme described in [20], as a high-throughput entropy encoder in a high frame rate video format [21], or in general as an entropy encoder in schemes for sparse coding [22], learned image compression [23], compressive sensing [24], or point cloud data compression [25]. In [26], ANS is employed to code large-alphabets sources.

This work was supported in part by the Centre National d'Etudes Spatiales, and by the Spanish Ministry of Economy and Competitiveness (MINECO), by the European Regional Development Fund (FEDER) and by the European Union under grant RTI2018-095287-B-I00, and by the Catalan Government under grants Beatriu de Pinós BP2018-00008 and 2017SGR-463.

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org., provided by the author. The material contains the test frequency tables referenced in this document. This material is 304 KB in size."

While most of the interest in ANS from the data compression community has been focused on practical efforts, other relevant papers have been published recently, such as an approximate method of calculating the efficiency of encoders [27], [28] an improved probability approximation strategy [29], or their use as extremely parallel decoders [30].

In this manuscript, we study Tabled ANS (tANS), which is an ANS variant that is well suited for hardware implementations [31], [32], [33]. It employs a finite state machine that transitions from state to state as symbols are encoded, which is certainly not a new idea [34]. However, the ANS theory enables the creation of effective encoding tables for such encoders. The underlying idea of this ANS variant is that for non-integer amounts of information an encoder produces integer-length codewords, while state transitions are employed to carry over fractional bits of information to subsequent codewords.

In particular, we focus our study on efficiently measuring the redundancy produced by tANS encoders, and on optimizing tANS encoders to minimize their implementation footprint or to improve their coding efficiency. We introduce a novel redundancy measurement strategy that exploits the structure imposed by tANS encoders in state transition matrices. This strategy achieves almost-linear time complexities for redundancy calculations when combined with an efficient calculation of average codeword length for all states. In addition, we employ the proposed redundancy calculation technique to optimize tANS automatons. We do so by exploring automaton permutations through a stochastic hill climbing method. We expect these efforts to translate into, for example, less area being allocated to a hardware implementation, to having smaller coding tables sent as side information, or to compression methods that can dynamically determine adequate encoder sizes. Many compression techniques incorporate entropy encoders with suitably high efficiency. In this work, we aim at providing better tANS encoders at equivalent efficiency levels, but with lower hardware costs. Particularly relevant to this research is the optimization strategy for tANS encoders in [35]. Further comparison with this method is provided in the following sections.

The rest of this paper is organized as follows. In what remains of this introduction we present some background information on tANS. Afterward, we provide an efficient method to calculate the redundancy of tANS encoders in Section II, and we show a method to optimize tANS encoders in Section III. In Section IV we discuss our experimental results, and conclusions are drawn in Section V.

A. Tabled Asymmetric Numeral Systems

In this section we provide the necessary background on how tANS encoders operate and establish some common notation. For the underlying rationale behind tANS see [2]. See Table V for a description of each symbol.

Let an n-symbol discrete memoryless source have an underlying alphabet $\mathcal{A} = \{0, \ldots, n-1\}$. For symbol $s \in \mathcal{A}$, let p_s be the associated occurrence probability, which we assume to be finitely representable (otherwise, see [29]).

A tANS encoder (or decoder) of size m is a deterministic finite-state automaton that transitions from state to state as symbols are being encoded (or decoded). Such automaton is defined by its *key* vector, $\mathbf{f} = f_0 f_1 \ldots f_{m-1} \in \mathcal{A}^m$, which uniquely describes the encoding and decoding functions that the automaton employs (i.e., how are symbols mapped to codewords and which state transitions occur). Let m_s be the number of occurrences of symbol s in $\mathbf{f}$, with $\sum_{s \in \mathcal{A}} m_s = m$. For example, given $\mathcal{A} = \{0, 1, 2\}$, the key $\mathbf{f} = 00210$ defines an automaton of size $m = 5$, with $m_0 = 3$ and $m_1 = m_2 = 1$.

Given the current state of the automaton, represented as an integer $x \in I = \{m, \ldots, 2m-1\}$, the symbol s is encoded as follows:

1) First, state x is renormalized to $x' \in I_s = \{m_s, \ldots, 2m_s - 1\}$ by removing as many least significant bits from x as necessary (i.e., a bitwise right shift operation), and pushing them to a stack $\mathfrak{s}$ (least-significant bit first).
2) Then, an encoding function $C_s : I_s \to I$ is employed to produce the resulting state $C_s(x') = y$.

Hence, the encoding process results in a state transition from $x \in I$ to $y \in I$ and some bits pushed onto a stack.

The resulting compressed file contains only the contents of the stack, the final state of the automaton, and, if it cannot be deduced, the number of symbols encoded. The initial state of the automaton need not be included in the compressed file, and can be chosen arbitrarily. However an initial state $x = m$ is expected to produce less renormalization bits for the first encoded symbol (renormalizing any initial state larger than m produces at least as many least-significant bits and possibly more).

Decoding operations are performed in reverse order, with the peculiarity that the sequence of decoded symbols is produced in reversed order. To decode a symbol, a decoding function $D : I \to (\mathcal{A}, I_s)$ is employed to produce a decoded symbol s and a state $x' \in I_s$. Then, state x' is renormalized by popping as many bits as necessary from the stack $\mathfrak{s}$ and appending them as least-significant bits to x' so that the resulting x is in I.

Algorithms 1 and 2 below describe the procedure to encode and decode one symbol, respectively. The encoding and decoding functions are uniquely determined by the key $\mathbf{f}$ as discussed in more detail in what follows.

Algorithm 1 Algorithm for encoding one symbol.

```
Input: $s$, $x$, $\mathfrak{s}$, $m_s$, $C_s$
Output: $y$, $\mathfrak{s}$
  $x' \leftarrow x$
  while $x' > 2m_s - 1$ do
      push($x'$ mod $2, \mathfrak{s}$)
      $x' \leftarrow \lfloor x'/2 \rfloor$
  end while
  $y \leftarrow C_s(x')$
```

Algorithm 2 Algorithm for decoding one symbol.

```
Input: $y$, $\mathfrak{s}$, $m$, $D$
Output: $s$, $x$, $\mathfrak{s}$
  $(s, x') \leftarrow D(y)$
  while $x' < m$ do
      $x' \leftarrow 2x' +$ pop($\mathfrak{s}$)
  end while
  $x \leftarrow x'$
```

As an example, the encoding and decoding functions for the automaton with key $\mathbf{f} = 10211011$ are provided in Table I. For this key, $m_0 = 2$, $m_1 = 5$, $m_2 = 1$ and $m = 8$. To encode symbol $s = 0$ in this example, suppose that the current state of the automaton is $x = 8$. Renormalizing x into $x' \in I_0 = \{m_0, \ldots, 2m_0 - 1\} = \{2, 3\}$ requires taking the two least significant bits from x, resulting in $x' = 2$. Applying C_0 to x' yields $y = 9$, which is the new state of the automaton. The two least significant bits "00" are pushed to the stack.

The decoder starts with state $y = 9$ and applies D to obtain $s = 0$ and $x' = 2$. Renormalizing x' by popping bits from the stack and appending until $x \in I = \{m, \ldots, 2m-1\} = \{8, \ldots, 15\}$ yields $x = 8$.

A key $\mathbf{f}$ unambiguously defines functions C_s and D as follows:

- The first element of the ordered pair produced by the decoding function is obtained directly from the symbols, in order, contained in $\mathbf{f}$. That is,
$$D(x) = (f_{x-m}, x'), \; m \le x < 2m. \tag{1}$$
The second element of the ordered pair, x', is given after the definition for C_s below.
- The values in the coding table for C_s are, in order, the states that have the symbol s as the first element of the ordered pair in the coding table for D.
- Destination states in D (the second element of the ordered pair) are inverse to those in C_s. I.e., $D(x) = (s, x')$ where $x = C_s(x')$.

Table I: Encoding and decoding tables for the automaton with key '10211011'.

x	C_0	C_1	C_2	D
1			10	
2	9			
3	13			
4				
5		8		
6		11		
7		12		
8		14		$(1,5)$
9		15		$(0,2)$
10				$(2,1)$
11				$(1,6)$
12				$(1,7)$
13				$(0,3)$
14				$(1,8)$
15				$(1,9)$

It is well known that low encoding redundancies are obtained for keys where $m_s \approx m\,p_s$ [2]. However, as others have shown [35] and we further show, lower redundancies can be achieved by taking into account the order of symbols in $\mathbf{f}$. An efective key construction method is described in [2], which we use as a baseline. We reproduce that method in Algorithm 3 with slight modifications to ensure that all symbols appear at least once in the key. In the algorithm, heap tuples are compared lexicographically, and the 'pop' operation returns the smallest element.

Algorithm 3 Algorithm for baseline key creation.

```
Input: $m$, $p_s\,\forall s \in \mathcal{A}$
Output: $\mathbf{f}$
  $\mathfrak{g} \leftarrow$ Set $\{0, \dots, n-1\}$
  $\mathfrak{h} \leftarrow$ Heap $\{(0.5/p_0, 0), \dots, (0.5/p_{n-1}, n-1)\}$
  for $i = 0$ to $m-1$ do
      repeat
          $(v, s) \leftarrow$ pop($\mathfrak{h}$)
      until $s \in \mathfrak{g}$ or $|\mathfrak{g}| < m - i$
      push($(v + 1/p_s, s), \mathfrak{h}$)
      remove($s, \mathfrak{g}$)
      $f_i \leftarrow s$
  end for
```

II. Efficient Redundancy Calculation

As for any entropy encoder, the redundancy of a tANS encoder for a given source can be obtained as the difference between the entropy of the source and the average codeword length produced by the encoder. Thus, less redundant encoders produce smaller compressed files.

While redundancy calculation for tANS is well understood, the straightforward calculation of average codeword lengths is a computationally expensive procedure, which prevents its use in important applications. In particular, straightforward calculation can become infeasible for large automatons, empirical redundancy studies, and most importantly, data-driven key optimization procedures.

It is worth noting that a redundancy approximation to tANS was provided by Duda [2] and later refined by Yokoo [28]. However, the approximations may present notable divergences from the true redundancy or may fail to distinguish the best of several tANS encoders. See Fig. 1 for an example where Duda's redundancy approximation yields the same result for two different automatons, and where redundancy is significantly underestimated around $p_1 \simeq 0.2$. As seen in the figure, the same statement is true for Yokoo's approximation.

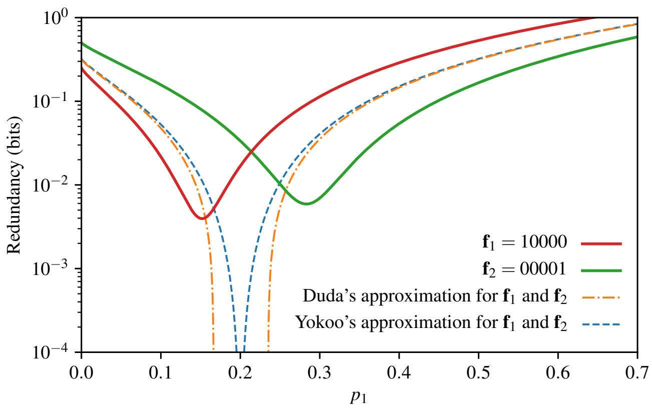


Figure 1: Example where Duda's and Yokoo's redundancy approximations yield identical results for two different automatons having f_1 and f_2, which in reality have significantly different redundancy profiles. Additionally, both methods dramatically underestimate redundancy for $p_1 \simeq 0.2$.

In this section we describe a procedure to efficiently calculate average codeword lengths and, in turn, the redundancy of tANS encoders.

The average codeword length of a tANS encoder can be written as

$$L = \sum_{x \in I} \mathcal{P}(x) \cdot \ell_x \tag{2}$$

where $\mathcal{P}(x)$ is the probability of the automaton being in the state x in a given instant, and ℓ_x is the per-state average codeword length.

State transitions can be modeled as a Markov process for which $\mathcal{P}(x)$ is the stationary probability of being in state x. These probabilities can be obtained as the dominant unitary left eigenvector of the transition matrix $\mathbf{P}$ (i.e., the dominant unitary right eigenvector of $\mathbf{P}^{\mathrm{T}}$). Here we consider only irreducible aperiodic Markov chains, so that unique stationary probabilities are guaranteed to exist. In particular, stationary probabilities for automatons with reducible Markov chains may not be unique, i.e., there can be more than one stationary distribution for a given automaton. On the other hand, a stationary probability may fail to exist for a periodic Markov chain. While outside the scope of this article, if necessary, an approximate solution can be found for these cases by adding a small constant to all the elements of the stochastic matrix and thus ensuring irreducibly and aperiodicity [36].

Regarding per-state average codeword lengths, these can be obtained as

$$\ell_x = \sum_{s\in\mathcal{A}} p_s \widehat{C}(s,x), \tag{3}$$

where $\widehat{C}(s,x) = \left\lfloor \lg \frac{x}{m_s} \right\rfloor$ is the number of bits pushed to the stack by Algorithm 1 when encoding symbol s in state x. Here, and in what follows, note that lg denotes the base-two logarithm. It is well known that, for each symbol s, $\widehat{C}(s,x)$ can only take two values, which are consecutive integers, and that a threshold t_s on x suffices to distinguish the correct value [2], [35]. Specifically,

$$\widehat{C}(s,x) = \begin{cases} k_s - 1 & x < t_s, \\ k_s & x \geq t_s, \end{cases} \tag{4}$$

where $k_s = \lfloor \lg(m/m_s) \rfloor + 1$ and $t_s = m_s 2^{k_s}$.

In what follows we show how to obtain ℓ_x and $P(x)$ in a computationally efficient manner, to facilitate the calculation of Eq. 2 (as illustrated in Fig. 2). First we describe how to calculate per-state average codeword lengths directly from the automaton size, symbol occurrence counts in the automaton key, and symbol probabilities. Afterwards, we describe how to obtain state probabilities by exploiting the structure of the state transition matrix of a tANS encoder. We obtain a compact representation this matrix, which we then use in a modified power method to obtain its dominant eigenvector.

A. Per-state average codeword length

While a straightforward approach to calculate all ℓ_x values through (3) and (4) requires $O(mn)$ operations, we show a method that obtains per-state average codeword lengths in $O(m)$ by employing finite differences. Note that throughout the document we assume that arithmetic operations have a $O(1)$ complexity, including exponentiation and logarithms.

Applying a forward difference operator (i.e., $\Delta_n a_n = a_{n+1} - a_n$) to ℓ_x yields

$$\Delta_x \ell_x = \sum_{s\in\mathcal{A}} p_s \Delta_x \widehat{C}(s,x), \tag{5}$$

where

$$\Delta_x \widehat{C}(s,x) = \begin{cases} 1 & \text{for } x = m_s 2^{k_s}, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Given symbol s, the value of $\Delta_x \widehat{C}(s,x)$ is only 1 for a single x value, which implies that each term in the summation in (5) is only non-zero for a single x value. Given this fact, we can obtain all values of $\Delta_x \ell_x$ at once by iterating over alphabet symbols and only calculating non-zero terms in the summation. Values of ℓ_x can then be found through cumulative summation from $\Delta_x \ell_x$ values, except for ℓ_m, which needs to be obtained from (5) directly. This method is formalized in Algorithm 4.

Regarding the complexity of the algorithm, given that $|\mathcal{A}| \leq |I|$, it can be seen that complexity is dominated by the first and last loops. Each loop performs m individual operations, and thus the algorithm complexity is $O(m)$.

Algorithm 4 Per-state average codeword length calculation.

```
Input: $m$, $m_s, p_s \,\forall s \in \mathcal{A}$
Output: $\ell_x \,\forall x \in I$
  for $x \in I$ do
      $\ell'_x \leftarrow 0$
  end for
  for $s \in \mathcal{A}$ do
      $k \leftarrow \lfloor \lg(m/m_s) \rfloor + 1$
      $t \leftarrow m_s 2^k$
      $\ell'_t \leftarrow \ell'_t + p_s$
      $\ell'_m \leftarrow \ell'_m + p_s(k-1)$
  end for
  $\ell_m \leftarrow \ell'_m$
  for $x = m+1$ to $2m-1$ do
      $\ell_x \leftarrow \ell_{x-1} + \ell'_x$.
  end for
```

B. Efficient calculation of the state transition matrix and its dominant eigenvector

Having already seen how to obtain ℓ_x, in this section we proceed to obtain $\mathcal{P}(x)$ by exploiting the particular structure of the transition matrix $\mathbf{P}$ of tANS automatons.

The re-normalization process in tANS creates structure and regularity in the automaton transition matrix which we can exploit. Due to re-normalization it can easily be seen that, in Algorithm 1, multiple and consecutive input values of x produce the same output value (same state transition) due to the floor operation applied in the re-normalization step.

For example, we show in Fig. 3 the transition matrix $\mathbf{P}$ and key $\mathbf{f}$ for a three-symbol automaton assuming $p_0 = 0.2$, $p_1 = 0.3$, and $p_2 = 0.5$. It is readily apparent that row i of $\mathbf{P}^{\mathrm{T}}$ is formed by runs of value p_{f_i}, which occasionally wrap around. I.e., runs containing the probability of the symbol in the same row of $\mathbf{f}^T$, which occasionally reach the right-most column continue on the left-most column. We can employ this fact to efficiently create equivalent compact representations of the transition matrices of tANS automatons. In addition, it is possible to observe that runs for the same symbol are successive and of equal length, up to the wrapping point, where its length is halved, but we do not exploit this for our purposes.

We can represent row $y-m$ of $\mathbf{P}^{\mathrm{T}}$, associated with destination state $y \in I$ as $M_y = (\alpha, \beta, p)$, where the interval $[\alpha, \beta)$ specifies a run of origin states $x \in I$ that lead to y with probability p. For runs that wrap around, the end of the run β is increased by m so that $\alpha < \beta$, which simplifies notation in further steps. In the example of Fig. 3, there are 15 states labeled 15 through 29. We then have $M_{15} = (20, 22, 0.2)$ for the top row (row 0) of $\mathbf{P}^{\mathrm{T}}$, and $M_{27} = (28, 31, 0.3)$

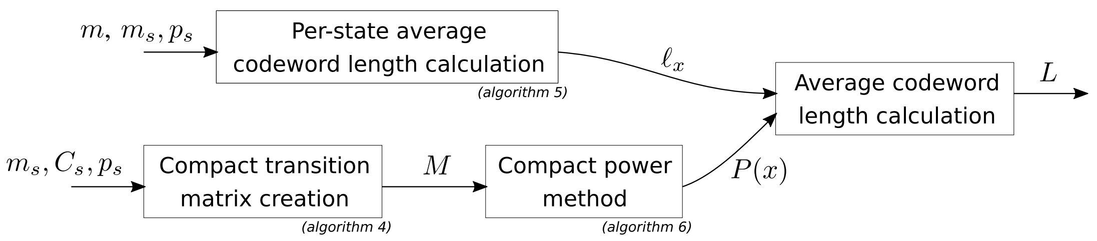

$m,\ m_s, p_s$
Per-state average codeword length calculation
(algorithm 5)
ℓ_x
Average codeword length calculation
L
m_s, C_s, p_s
Compact transition matrix creation
(algorithm 4)
M
Compact power method
(algorithm 6)
$P(x)$

Figure 2: Diagram of the redundancy calculation method.

$$\mathbf{P}^{\mathrm{T}} = \begin{pmatrix}
 & & & & & 0.2 & 0.2 & & & & & & & & \\
 & & & & & & & 0.2 & 0.2 & & & & & & \\
 & & & & & & & & & 0.2 & 0.2 & & & & \\
 & & & & & & & & & & & 0.2 & 0.2 & & \\
 & 0.3 & 0.3 & 0.3 & 0.3 & & & & & & & & & & \\
 & & & & & 0.3 & 0.3 & 0.3 & 0.3 & & & & & & \\
 & & & & & & & & & & & & & 0.2 & 0.2 \\
0.2 & & & & & & & & & & & & & & \\
 & 0.2 & & & & & & & & & & & & & \\
 & & 0.2 & & & & & & & & & & & & \\
 & & & 0.2 & & & & & & & & & & & \\
 & & & & & & & & & 0.3 & 0.3 & 0.3 & 0.3 & & \\
0.3 & & & & & & & & & & & & & 0.3 & 0.3 \\
 & & & & 0.2 & & & & & & & & & & \\
0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5
\end{pmatrix}, \quad \mathbf{f}^T = \begin{pmatrix} 0\\0\\0\\0\\1\\1\\0\\0\\0\\0\\0\\1\\1\\0\\2 \end{pmatrix}$$

Figure 3: Transition matrix (transposed) and key for an automaton with $n = 3$, $p_0 = 0.2$, $p_1 = 0.3$, $p_2 = 0.5$, and $m = 15$.

for the third row from the bottom (row 12). We can find all states $x \in I$ with destination state $y \in I$ starting from the re-normalized state $x' \in I_s$ such that $C_s(x') = y$ as follows. We look for all $x \in I$ that re-normalize to x', i.e., $x = x'2^k + z$, with $0 \leq z \leq 2^k - 1$ and $k, z \in \mathbb{Z}$. For $z = 0$ we obtain a run start by solving $x'2^k + 0 \geq m$, which yields $k = \lceil \lg m/x' \rceil$, as only one value for k is possible [2]. Similarly, for $z = 2^k - 1$ we obtain a run end by solving $x'2^k + (2^k - 1) + 1 \geq m$, which yields $k = \lceil \lg m/(x'+1) \rceil$. Hence, $\alpha = x'2^{\lceil \lg m/x' \rceil}$, and either $\beta = (x'+1)2^{\lceil \lg m/(x'+1) \rceil}$ or $\beta = (x'+1)2^{\lceil \lg m/(x'+1) \rceil} + m$, depending whether there is a wrap around or not. This yields $M_y = (\alpha, \beta, p)$.

Hereafter we refer to this representation, M, as a compact transition matrix, which Algorithm 5 obtains in $O(m)$ time complexity, given that $|\bigcup_{s \in \mathcal{A}} I_s| = m$.

Algorithm 5 Compact transition matrix creation.

```
Input: $p_s, m_s, C_s \ \forall s \in \mathcal{A}$
Output: $M$
  for $s \in \mathcal{A}$ do
      for $x' \in I_s$ do
          $\alpha \leftarrow x' \cdot 2^{\lceil \lg(m/x') \rceil}$
          $\beta \leftarrow (x'+1) \cdot 2^{\lceil \lg(m/(x'+1)) \rceil}$
          $\beta \leftarrow \begin{cases} \beta & \text{if } \alpha < \beta \\ \beta + m & \text{otherwise} \end{cases}$
          $M_{C_s(x)} = (\alpha, \beta, p_s)$
      end for
  end for
```

Now we see that we can employ the power method on a compact transition matrix to produce its dominant eigenvector (i.e., the vector with elements $\mathcal{P}(x)\ \forall x \in I$). It is well known that the power method can be employed by iterating over

$$\mathbf{v_{i+1}}^T = \frac{\mathbf{P}^{\mathrm{T}}\mathbf{v_i}^T}{||\mathbf{P}^{\mathrm{T}}\mathbf{v_i}^T||} \tag{7}$$

to produce a dominant eigenvector in $O\left(\frac{R}{-\lg|\lambda_2|}\right)$ iterations, where R is the required precision and λ_2 is the second-largest eigenvalue [37] (see [38] for further estimation of this eigenvalue). In what follows, we show that, by exploiting the transition matrix structure, we can obtain the dominant eigenvector in a $O\left(m\frac{R}{-\lg|\lambda_2|}\right)$ complexity. To do so, we show how to obtain the result of the matrix-vector multiplication $\mathbf{w}^T = \mathbf{P}^{\mathrm{T}}\mathbf{v}^T$ in the power method in $O(m)$ instead of $O(m^2)$.

For non-wrapping runs, the i^{th} element of $\mathbf{w}^T$ can be formulated as

$$w_i = p_{f_i} \sum_{\alpha \leq j < \beta} v_{j-m}, \tag{8}$$

with α and β being the closed and open boundaries of the run in row i of the transition matrix, as described previously. For wrapping runs we can consider

$$w_i = p_{f_i} \left(\sum_{\alpha \leq j < 2m} v_{j-m} + \sum_{m \leq j < \beta - m} v_{j-m} \right). \tag{9}$$

Due to the particular definition of β, both summations in (9) are equivalent to

$$w_i = p_{f_i} \left(\sum_{\alpha \le j < \beta} u_{j-m} \right) , \qquad (10)$$

where $\mathbf{u} = (v_0, \ldots, v_{m-1}, v_0, \ldots, v_{m-1})$. Thanks to this, arbitrary summations of continuous u_i elements can be obtained in $O(1)$ by subtracting two elements from a pre-calculated cumulative sum of the elements in $\mathbf{u}$.

The resulting method to obtain stationary state probabilities for tANS automatons is presented in Algorithm 6. As previously discussed, Algorithm 6 converges in $O\left(\frac{R}{-\lg|\lambda_2|}\right)$ iterations, with a $O(m)$ cost per iteration. Thus the complexity of Algorithm 6 is $O\left(m\,\frac{R}{-\lg|\lambda_2|}\right)$. This complexity dominates those from the previous steps and results in the total complexity for the tANS redundancy calculation.

Algorithm 6 Compact power method.

```
Input: $m, M$
Output: $\mathbf{v}$
  $\mathbf{v} \leftarrow \frac{\mathbf{1}}{||\mathbf{1}||_2}$
  repeat
    $\mathbf{v}' \leftarrow \mathbf{v}$
    $u'_0 \leftarrow 0$
    for $i \leftarrow 1$ to $2m-1$ do
      $u'_i \leftarrow u'_{i-1} + v_{(i-1 \mod m)+1}$
    end for
    for $i \leftarrow 0$ to $m-1$ do
      $(\alpha, \beta, p) \leftarrow M_{i+m}$
      $v_i \leftarrow p \cdot (u'_\beta - u'_\alpha)$;
    end for
    $\mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||_2}$
  until $||\mathbf{v} - \mathbf{v}'||_\infty < 2^{-R}$
```

At this point we have presented a method to obtain the average codeword length of a tANS automaton and thus its redundancy in a very efficient manner, which allows us to explore the optimization of automatons.

III. Redundancy reduction

The basic principle behind ANS dictates that, when encoding a symbol, an encoder should transition to a next state that is approximately $1/p_s$ times larger than the current state (i.e., $y \approx x/p_s$), in order to accommodate the extra $\lg(1/p_s)$ bits of information [2]. When this is translated to tANS, it suggests that the number of occurrences of each symbol in tANS keys should be $m_s \approx m p_s$. However, in addition to symbol frequencies, the order of these symbols in the key also plays an important role in tANS performance, as larger states emit more symbols to reach the same I_s during re-normalization, and not all automaton states are equally probable [2].

For example, Fig. 4 shows the redundancy of all two-symbol automatons that have keys sorted in ascending order, which we call ordered automatons. There are $m{+}1$ such automatons. The 26 dotted black curves in the figure correspond to all ordered automatons with $m = 25$. The blue curve is the minimum of these. The other colored curves show the minimum curve for other values of m. For comparison, the black curve represents the redundancy obtained by non-ordered automatons produced by Algorithm 3 for $m = 100$. It can be seen that, as m increases, the redundancies obtained by even very large ordered automatons seem to stagnate, still having relative redundancies significantly larger than the automatons with non-ordered symbols. Note how the curves for $m = 1000$, $m = 10000$, and $m = 100000$ are nearly identical. This might have interesting implications for the redundancies achievable by the rANS variant of ANS, which for redundancy purposes are equivalent to ordered tANS automatons, as it seems to suggest that rANS encoders may not converge to zero redundancy as automatons grow, or that they do so at a much slower rate than non-ordered tANS automatons.

[1]This figure required more than 50 million redundancy evaluations for automatons of 10^5 states. Calculation was possible thanks to the redundancy calculation method presented in the previous section.

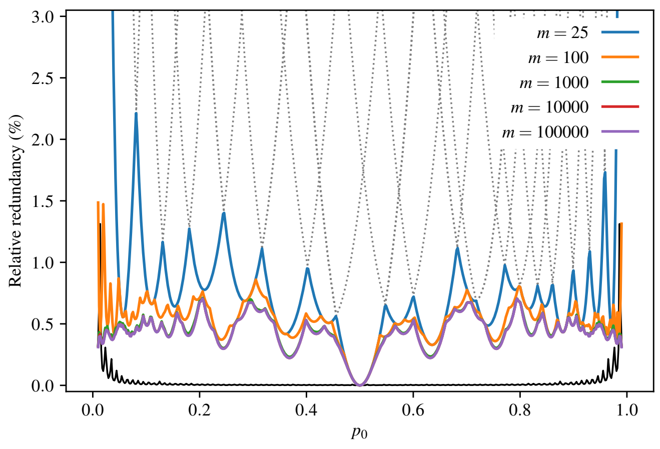


Figure 4: Redundancy achieved with automatons of a given size where keys are restricted to be in order.[1]

In the remainder of this section we describe how to obtain well-performing orders for the elements of tANS automaton keys. The approach, which is based on hill climbing, is as follows: given an initial automaton key, a succession of pseudo-random pairwise permutations of key elements are considered. Whenever an element permutation reduces the automaton redundancy, the permutation is applied to the key, otherwise the permutation is discarded. Algorithm 3 can be employed to create initial automaton keys that have appropriate symbol frequencies.

Traditionally, measuring the effects of any replacement or permutation of the key elements has been computationally challenging. In particular, obtaining the dominant eigenvector after a change in a state transition matrix is a difficult and well-studied problem [37], [39]. Existing solutions are significantly more computationally expensive than the straightforward application of our redundancy calculation method, or are just not very effective. For example, employing an approximate eigenvector as the starting point for the power method may seem promising, but is not as effective as intuition would suggest [37]. Hence, in what follows, we directly employ the proposed redundancy calculation method to evaluate the effect of a key change.

Informed decisions could also be employed to explore the space of automaton keys. For example, the estimator

$$\begin{aligned} E &= -\mathcal{P}(x)\ell_x - \mathcal{P}(y)\ell_y + \mathcal{P}(x)\ell_y + \mathcal{P}(y)\ell_x \\ &= (\mathcal{P}(x) - \mathcal{P}(y))(\ell_y - \ell_x) \end{aligned} \tag{11}$$

could be employed to guide the selection of keys to be evaluated. This estimator is the redundancy variation after the elements of a key associated with states x and y are swapped, under the assumption that state probabilities will remain unaltered. Recently, Dubé and Yokoo proposed a similar approach in [35], where key elements are rearranged as per the probability of their associated automaton state in descending order. In their state-of-the-art proposal this approach is repeated multiple times, and the key producing the least redundancy is finally selected.

The motivation of such approach is to reduce complexity by reducing the number of keys to be evaluated. Given the low cost of evaluating keys via the proposed method, such approaches may not be necessary. Indeed, directly exploring pseudo-random permutations reaches more key variations than those suggested by heuristics and approximations, and ultimately achieve lower redundancies.

Fig. 5 shows redundancy reductions versus iterations for three different frequency tables for our proposed method as compared to the informed approach proposed by Dubé and Yokoo. It is worth emphasizing that for both approaches the plots indicate actual redundancy reduction calculated via algorithms proposed earlier in this paper, without employing any approximations or heuristics. In each case, the key size m is set to approximately $5n$. It can be observed that the informed approach yields significant redundancy reductions with very few iterations, with little or no further improvement provided by subsequent iterations. In contrast, for pseudo-random permutations, significant redundancy reductions are only obtained after a moderate number of iterations, but a larger redundancy reduction is eventually obtained.

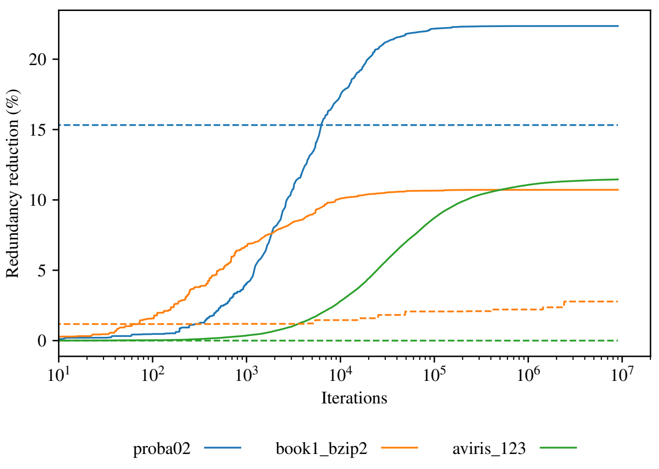


Figure 5: Redundancy reduction versus number of iterations for the proposed method (solid curves) and for the method by Dubé and Yokoo (dashed curves). Results are for the first frequency table of each dataset.

IV. Experimental Results

In this section we first present the corpus employed to test the described methods, followed by experimental results regarding the performance of the method proposed in Section II, and results regarding the optimization method described in Section III.

In order for the experimental results to be applicable in practical scenarios, a data corpus has been carefully curated. The corpus, detailed in Table II, consists of multiple frequency tables obtained in various real data compression experiments.[2] Two of the corpus datasets (aviris_123 and deer_iwt) represent what an entropy encoder may expect during image compression, whereas another two (book1_bzip2, enwik8_gzip) are representative of the expected input to the entropy encoder of a general-purpose compression method. Three additional synthetic distributions are included in the corpus.

Regarding the performance of the method presented in Section II, in addition to the previous theoretical time complexity discussion, several wall-clock measurements are reported here. Table III reports on performance of the proposed redundancy calculation method. Results are produced on an AMD Threadripper 1950X with 3200 Mhz DDR4 RAM, by employing a hybrid implementation of higher level Python 3 code and lower-level elements either in NumPy or custom C code. All results are for single-thread execution (including BLAS routines). We have also developed a 'reference' implementation for comparison purposes where the power method is employed, but none of the other complexity reduction strategies described in Section II are used. As compared with this reference implementation, the proposed method yields substantial execution time improvements. Albeit the reference implementation is not as optimized as the proposed one, the difference in wall-clock performance is of around three or four orders of magnitude. For $m \geq 10^5$, results for the reference method have not been produced due to its quadratic memory requirements. In addition, the proposed technique has been checked against this reference implementation for correctness, with additional checks against LAPACK routines for the power method.

Regarding the performance of the optimization method presented in Section III, results are presented in Fig. 6 and Table IV. Results in Fig. 6 show the relation between automaton size and its redundancy for two different frequency tables. Results for a 'default' automaton (Algorithm 3) are compared with the proposed 'optimized' method, and with three random permutations of the 'default' automaton. It can be observed that random permutations tend to yield poor results, whereas the optimized method provides improvements over the default

[2]The corpus is available online as supplementary material through IEEE Xplore.

Table II: Details of the corpus of frequency tables employed in the experimental results. The minimum, average, and maximum number of symbols is reported for each set of frequency tables.

Name	Tables	Min. n	Avg. n	Max. n	Source
aviris_123	11	355	1,272.1	4,891	CCSDS 123.0-B-2 applied to the AVIRIS Yellowstone hyperspectral image (scene 00).
book1_bzip2	24	92	92.0	92	Bzip2 (-9) on the ‘BOOK1’ file from the Calgary corpus.
deer_iwt	16	3,412	9,395.2	14,447	Subbands of a 5-level 5/3 LeGall IWT transform applied to the ‘deer’ image from the 16-bit (log) Rawzor corpus.
enwik8_gzip	1,985	13	82.1	219	Gzip (-9) on M. Mahoney’s 100 MB English Wikipedia test file.
proba02	1	256	256.0	256	Synthetic distribution (02) from Y. Collet’s ‘probaGenerator’.
proba14	1	53	53.0	53	Synthetic distribution (14) from Y. Collet’s ‘probaGenerator’.
proba80	1	7	7.0	7	Synthetic distribution (80) from Y. Collet’s ‘probaGenerator’.

Table III: Wall-clock measurements of the proposed method (measured in seconds). The first table of each dataset is employed for the measurements. Invalid automaton sizes ($m < n$) are denoted by ‘-’.

	Reference			Proposal			Speedup		
m	book1_bzip2	deer_iwt	proba02	book1_bzip2	deer_iwt	proba02	book1_bzip2	deer_iwt	proba02
10^2	0.152	-	-	0.001	-	-	152.0	-	-
10^3	1.723	-	0.496	0.002	-	0.002	861.5	-	248.0
10^4	32.940	116.316	5.479	0.011	0.014	0.010	2994.5	8308.3	547.9
10^5	Insufficient memory			0.130	0.155	0.099	Insufficient memory		
10^6				3.614	9.409	1.891			
10^7				39.588	191.075	23.173			

automaton.

For some frequency tables, such as that in Fig. 6(a), larger automatons consistently yield smaller redundancies. However, for the frequency table presented in Fig. 6(b) and others, the redundancy obtained by default automatons may significantly increase as their size is increased. Thus, even for cases where automaton optimization may not be worth the extra computational effort, carefully evaluating automaton size can yields important benefits.

In Table IV, comprehensive results are presented for all frequency tables of all data sets. It is worth mentioning that automaton keys that are close in size to the number of alphabet symbols may present less opportunities to improve performance by permuting key elements (e.g., the improvement obtained for the proba02 table is negligible for $m \simeq 1.1n$). However, these keys may also fail to mimic the frequency table. This is particularly relevant for the highly-biased table in the proba80 data set, where an automaton with $m \simeq 1.1n = 7.7$ cannot effectively represent the required occurrence probabilities. However, as automaton sizes are increased, redundancy reductions become substantial, and even more so as compared to randomly selected permutations of an automaton key.

V. Conclusions

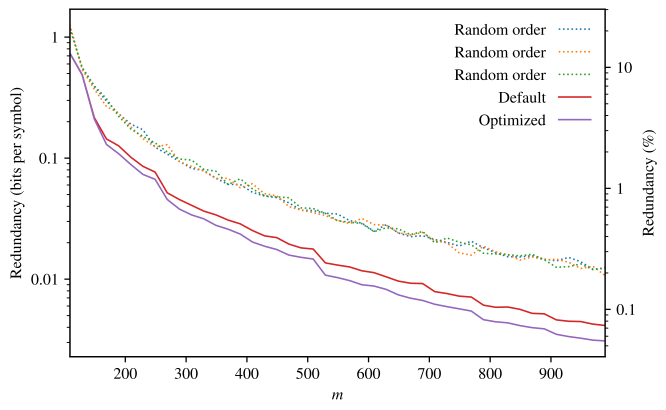
This paper addresses the problem of efficiently obtaining tANS automaton redundancies and the optimization of said automatons. We describe a method to efficiently obtain automaton redundancies, and we do so by efficiently obtaining per-state average codeword lengths through a forward difference operator, and by exploiting the structure of the state transition matrix to obtain the stationary probabilities of the automaton states. Our proposed method operates in almost-linear time complexity $O\left(m \frac{R}{-\lg|\lambda_2|}\right)$, and three to four orders of magnitude faster in practice. In addition, we show that we can employ the proposed redundancy calculation method in a stochastic hill climbing optimization process to improve tANS encoder designs. Experimental results with novel corpus of frequency tables obtained from realistic data compression processes indicate that automatons are consistently improved over other informed strategies of automaton optimization. In particular, results show that improvements increase as automatons grow larger.

VI. Acknowledgments

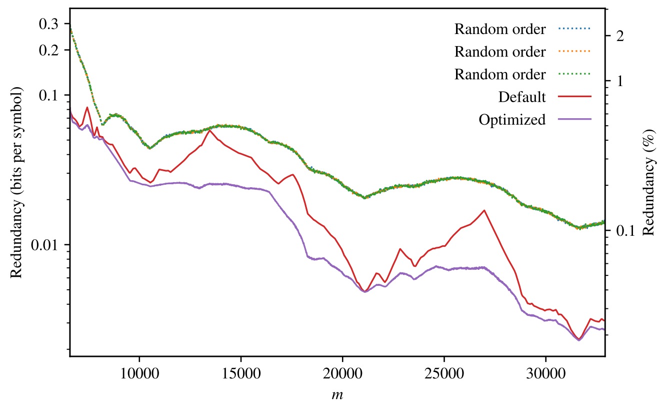
We would like to thank Charles Bloom for his well-thought-out “rants” blog.

Table IV: Redundancy reductions, in percentage, over default and random key construction methods. Automaton sizes are set as a multiple of the alphabet size. Results are averaged over all frequency tables in each data set. For each result 50 thousand iterations are used.

	$m \simeq 1.1n$		$m \simeq 2n$		$m \simeq 5n$	
Corpus	Default	Random	Default	Random	Default	Random
aviris_123	0.35	17.05	5.08	29.43	7.63	31.87
book1_bzip2	0.52	23.70	10.11	35.17	11.60	39.12
deer_iwt	2.17	41.33	17.40	72.66	17.04	90.83
enwik8_gzip	5.42	49.32	28.96	76.26	37.04	89.59
proba02	0.00	22.59	10.96	38.88	21.80	66.41
proba14	2.79	26.56	9.19	40.12	11.45	51.11
proba80	0.00	0.00	0.00	45.17	5.36	28.01



(a) First table in 'enwik8_gzip'.



(b) First table in 'deer_iwt'.

Figure 6: Redundancy as a function of automaton size.

APPENDIX

A. Notation

A summary of the symbols employed through this manuscript is included in Table V.

REFERENCES

[1] J. Duda, "Asymmetric numeral systems," *arXiv preprint arXiv:0902.0271*, 2009.

Table V: Symbol list.

Symbol	Description
$\mathcal{A}$	An alphabet
α, β	Beginning and end of a run in M_y
C_s	Coding function for symbol s
$\widehat{C}$	Bits pushed into the stack (function)
D	Decoding function
Δ_n	Forward difference operator
$\mathbf{f}$	Automaton key
f_i	i'th element in $\mathbf{f}$
$\mathfrak{g}$	A set
$\mathfrak{h}$	A heap
I	Interval $\{m, \ldots, 2m-1\}$
I_s	Interval $\{m_s, \ldots, 2m_s-1\}$
k_s	Threshold value in $\widehat{C}$ for symbol s
L	Automaton average codeword length
ℓ_x	Average codeword length for state x
λ_2	Second largest eigenvalue
M	Compact transition matrix
M_y	Run of origin states that lead to state y in a compact transition matrix
m	Automaton size
m_s	Number of occurrences of s in $\mathbf{f}$
n	Alphabet size
$\mathbf{P}$	Probability transition matrix
$\mathcal{P}(x)$	Probability of automaton being in state x
p_s	Occurrence probability of symbol s
R	Precision value
s	A symbol in $\mathcal{A}$
$\mathfrak{s}$	A stack
$\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$	Vectors employed in power method
u_i, v_i, w_i	i'th elements in $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, respectively
x, x', y	Automaton states

[2] ——, “Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding,” *arXiv preprint arXiv:1311.2540*, 2013.

[3] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, “The use of asymmetric numeral systems as an accurate replacement for Huffman coding,” in *2015 Picture Coding Symposium (PCS)*, May 2015, pp. 65–69.

[4] Y. Collet, “Finite State Entropy,” http://github.com/Cyan4973/FiniteStateEntropy, 2013.

[5] F. Giesen, “Interleaved Entropy Coders,” *arXiv preprint arXiv:1402.3392*, 2014.

[6] Y. Collet and C. Turner, “Smaller and faster data compression with Zstandard,” *Facebook Code*, 2016. [Online]. Available: https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/

[7] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, “Brotli: A general-purpose data compressor,” *ACM Transactions on Information Systems (TOIS)*, vol. 37, no. 1, pp. 4:1–4:30, 2018.

[8] J. Alakuijala and Z. Szabadka, “Brotli Compressed Data Format,” Internet Engineering Task Force, RFC 7932, July 2016.

[9] E. Bainville, “LZFSE,” https://github.com/lzfse/lzfse, 2017.

[10] A. Moffat and M. Petri, “ANS-based Index Compression,” in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, Feb. 2017, pp. 677–686.

[11] ——, “Index Compression Using Byte-Aligned ANS Coding and Two-Dimensional Contexts,” in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, Feb. 2018, pp. 405–413.

[12] F. Konstantinov, G. Gryzov, and K. Bystrov, “The Use of Asymmetric Numeral Systems Entropy Encoding in Video Compression,” in *Distributed Computer and Communication Networks*, V. M. Vishnevskiy, K. E. Samouylov, and D. V. Kozyrev, Eds. Cham: Springer International Publishing, 2019, pp. 125–139.

[13] J. Townsend, T. Bird, J. Kunze, and D. Barber, “HiLLoC: Lossless Image Compression with Hierarchical Latent Variable Models,” in *International Conference on Learning Representations*, May 2020, pp. 1–14.

[14] J. Townsend, T. Bird, and D. Barber, “Practical Lossless Compression with Latent Variables using Bits Back Coding,” in *International Conference on Learning Representations*, Apr. 2019, pp. 1–13.

[15] D. Marwood, P. Massimino, M. Covell, and S. Baluja, “Representing images in 200 bytes: Compression via triangulation,” in *ICIP 2018. Proceedings of 2018 International Conference on Image Processing*, 2018, pp. 405–409.

[16] P. A. Chou, M. Koroteev, and M. Krivokuća, “A volumetric approach to point cloud compression, part i: Attribute compression,” *IEEE Trans. Image Process.*, pp. 2203–2216, 2019.

[17] M. Krivokuća, P. A. Chou, and M. Koroteev, “A volumetric approach to point cloud compression, Part II: Geometry compression,” *IEEE Trans. Image Process.*, pp. 2217–2229, 2019.

[18] R. Long, M. Hernaez, I. Ochoa, and T. Weissman, “Genecomp, a new reference-based compressor for sam files,” in *2017 Data Compression Conference (DCC)*, 2017, pp. 330–339.

[19] P. Peter, “Fast inpainting-based compression: Combining shepard interpolation with joint inpainting and prediction,” in *ICIP 2019. Proceedings of 2019 International Conference on Image Processing*, 2019, pp. 3557–3561.

[20] Q. Chen, H. Sun, and N. Zheng, “Worst Case Driven Display Frame Compression for Energy-Efficient Ultra-HD Display Processing,” *IEEE Trans. Multimedia*, vol. 20, no. 5, pp. 1113–1125, May. 2018.

[21] A. Mackin, F. Zhang, and D. R. Bull, “A Study of High Frame Rate Video Formats,” *IEEE Trans. Multimedia*, vol. 21, no. 6, pp. 1499–1512, June 2019.

[22] M. Kalluri, M. Jiang, N. Ling, J. Zheng, and P. Zhang, “Adaptive RD Optimal Sparse Coding With Quantization for Image Compression,” *IEEE Trans. Multimedia*, vol. 21, no. 1, pp. 39–50, Jan. 2019.

[23] Z. Cheng, H. Sun, M. Takeuchi, and J. Katto, “Energy compaction-based image compression using convolutional autoencoder,” *IEEE Trans. Multimedia*, pp. 1–14, Aug. 2019 (early access).

[24] X. Yuan and R. Haimi-Cohen, “Image Compression Based on Compressive Sensing: End-to-End Comparison with JPEG,” *IEEE Trans. Multimedia*, pp. 1–16, Jan. 2020 (early access).

[25] P. de Oliveira Rente, C. Brites, J. Ascenso, and F. Pereira, “Graph-Based Static 3D Point Clouds Geometry Coding,” *IEEE Trans. Multimedia*, vol. 21, no. 2, pp. 284–299, Feb. 2019.

[26] A. Moffat and M. Petri, “Large-Alphabet Semi-Static Entropy Coding Via Asymmetric Numeral Systems,” *ACM Transactions on Information Systems*, vol. 38, no. 4, Jul. 2020.

[27] H. Yokoo, “On the Stationary Distribution of Asymmetric Binary Systems,” in *2016 IEEE International Symposium on Information Theory (ISIT)*, July 2016, pp. 11–15.

[28] ——, “On the Stationary Distribution of Asymmetric Numeral Systems,” in *2016 International Symposium on Information Theory and Its Applications (ISITA)*, Oct. 2016, pp. 631–635.

[29] H. Yokoo and T. Shimizu, “Probability Approximation in Asymmetric Numeral Systems,” in *2018 International Symposium on Information Theory and Its Applications (ISITA)*, Oct. 2018, pp. 638–642.

[30] A. Weiundefinedenberger and B. Schmidt, “Massively Parallel ANS Decoding on GPUs,” in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337888

[31] S. M. Najmabadi, Z. Wang, Y. Baroud, and S. Simon, “High Throughput Hardware Architectures for Asymmetric Numeral Systems Entropy Coding,” in *2015 9th international symposium on image and signal processing and analysis (ISPA)*, Sept. 2015, pp. 256–259.

[32] S. M. Najmabadi, H. S. Tungal, T.-H. Tran, and S. Simon, “Hardware-based architecture for asymmetric numeral systems entropy decoder,” in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Sept. 2017, pp. 1–6.

[33] S. M. Najmabadi, T.-H. Tran, S. Eissa, H. S. Tungal, and S. Simon, “An Architecture for Asymmetric Numeral Systems Entropy Decoder-A Comparison with a Canonical Huffman Decoder,” *Journal of Signal Processing Systems*, vol. 91, no. 7, pp. 805–817, 2019.

[34] M. J. Gormish and E. L. Schwartz, “Apparatus and method for performing m-ary finite state machine entropy coding,” Jun 1999, US Patent 5,912,636.

[35] D. Dubé and H. Yokoo, “Fast Construction of Almost Optimal Symbol Distributions for Asymmetric Numeral Systems,” in *2019 IEEE International Symposium on Information Theory (ISIT)*, July 2019, pp. 1682–1686.

[36] T. Haveliwala, S. Kamvar, D. Klein, C. Manning, and G. Golub, “Computing pagerank using power extrapolation,” Stanford InfoLab, Technical Report 2003–45, 2003.

[37] A. N. Langville and C. D. Meyer, “Updating Markov chains with an eye on Google’s PageRank,” *SIAM Journal on Matrix Analysis and Applications*, vol. 27, no. 4, pp. 968–987, 2006.

[38] S. T. Garren and R. L. Smith, “Estimating the second largest eigenvalue of a Markov transition matrix,” *Bernoulli*, vol. 6, no. 2, pp. 215–242, 2000.

[39] M. J. O’Hara, “On low-rank updates to the singular value and Tucker decompositions,” in *Proceedings of the 2010 SIAM International Conference on Data Mining*, 2010, pp. 713–719.