Efficiency of ANS Entropy Encoders

Dmitry Kosolobov

Abstract

Asymmetric Numeral Systems (ANS) is a class of entropy encoders that had an immense impact on the data compression, substituting arithmetic and Huffman coding. It was studied by different authors but the precise asymptotics of its redundancy (in relation to the entropy) was not completely understood. We obtain optimal bounds for the redundancy of the tabled ANS (tANS), the most popular ANS variant. Given a sequence $a_1, a_2, \ldots, a_n$ of symbols from an alphabet $\{0, 1, \ldots, \sigma - 1\}$ such that each symbol $a$ occurs in it $f_a$ times and $n = 2^k$, the tANS encoder using Duda’s “precise initialization” to fill tANS tables transforms this sequence into a bit string of the following length (the frequencies are not included in the encoding): 
$$\sum_{a \in \{0, \ldots, \sigma\}} f_a \cdot \log\frac{2}{f_a} + O(\sigma + r),$$
where $O(\sigma + r)$ can be bounded by $\sigma \log e + r$. The $r$-bit term is an artifact indispensable to ANS; the rest incurs a redundancy of $O\left(\frac{r}{n}\right)$ bits per symbol. We complement this example by showing that an $\Omega(\sigma + r)$ redundancy is necessary. We argue that similar examples exist for most adequate initialization methods for tANS. Thus, we refute Duda’s conjecture that the redundancy is $O\left(\frac{\sigma}{n}\right)$ bits per symbol. We also propose a variant of the range ANS (rANS), called rANS with fixed accuracy, parameterized by $k \geq 1$. In this variant the integer division, which is unavoidable in rANS, is performed only when its result belongs to $\{2^k \cdot 2^{k+1}\}$. Therefore, the division can be computed by faster methods provided $k$ is small. We bound the redundancy for our rANS variant by $\frac{\sigma}{2^{k+1}} \log e + r$.

Index Terms

Asymmetric numeral systems, ANS, finite state entropy, FSE, arithmetic coding, redundancy.

I. INTRODUCTION

ASYMMETRIC NUMERAL SYSTEMS (ANS) is a class of entropy encoders invented by Duda in 2009 \[13\], \[14\], \[16\]. These encoders had a huge impact on the data compression by providing the same rates of compression as the arithmetic coding \[23\], \[25\], \[29\] while being as fast as the Huffman coding \[20\] (and even faster in some scenarios). Since the invention of ANS and the emergence of its efficient implementation by Collet \[9\], several high performance compressors based on ANS appeared \[1\], \[8\], \[19\] and it was integrated in some modern media formats \[4\], \[5\]. The theoretical community also contributed to the study of ANS in a series of works \[6\], \[15\], \[26\], \[34\], \[35\], \[35\]–\[37\], though less actively than practitioners \[7\], \[10\], \[18\], \[19\].

The primary focus of the theoretical analysis for entropy encoders is in the estimation of the redundancy, the difference between the number of bits produced by the encoder and the information theoretic entropy lower bound. The results of the present paper are twofold: first, we establish tight asymptotic upper and lower bounds on the redundancy of the most popular variant of ANS, called the tabled ANS (tANS) or, sometimes, the Finite State Entropy (FSE); second, we introduce and analyze a novel variant of the range ANS (rANS), another version of ANS used in practice. Our analysis of the redundancy does not impose any randomized assumptions on the source that produced the input for the encoder; instead, we establish worst-case bounds in terms of the so-called empirical entropy \[21\], \[22\] (the definitions follow).

An entropy encoder receives as its input a sequence of $n$ numbers from a set $\{0, 1, \ldots, \sigma - 1\}$ and transforms it into a bit string. Duda analyzed his tANS encoder and found that its redundancy is $O\left(\frac{\sigma}{n} \log n\right)$ bits per symbol \[1\]; provided an appropriate initialization is used for tANS tables (details are explained in the sequel); see Equation (40) in \[13\]. His estimation, however, seems somewhat non-rigorous and in a different (more classical) setting: he considers a memoryless source generating the input at random and he estimates the expected length of the output encoding. Based on experimental evaluations, Duda conjectured that a tight bound for the redundancy is $O\left(\frac{\sigma}{n}\right)$ bits per symbol \[14\]. Yokoo and Dubé \[37\] investigated the same problem in more rigorous terms closer to our setting and showed that the redundancy per symbol vanishes as the length $n$ tends to infinity while $\sigma$ is fixed (yet, they have some questionable assumptions in their derivations). Another analysis of the expected output length for the memoryless source was conducted in \[35\]; their bounds, however, are incomplete in a sense and, thus, cannot be easily compared to other results. In contrast, our analysis establishes upper bounds in the worst case without probabilistic assumptions.

In this paper we prove that the redundancy for tANS is $O\left(\frac{\sigma}{n}\right)$ bits per symbol. We complement this upper bound by a series of examples showing that it is asymptotically tight when $\sigma > n/3$. As in the works cited above, here we did not include in this bound the $r$ redundant bits that are always produced by ANS encoders in the worst case (it is an artifact of the initial state of the encoder). After uncovering the constant under the big-O and including this $r$-bit term, the upper bound for the tANS redundancy that we establish can be expressed as $\sigma \log e + r$ bits over all symbols (not per symbol). Our lower bound

The author is with Ural Federal University, Ekaterinburg, Russia (e-mail: dkosolobov@mail.ru).

1All logarithms in the paper are in base two.
examples show that a redundancy of \(\frac{\sigma - 1}{\sigma} + r - 2\) bits is attainable, for \(\sigma \approx n/3\). An important part of tANS is the initialization of its internal tables, which has a significant impact on the compression performance. The lower bound examples work for a wide class of initialization algorithms so that the same redundancy can be observed for any adequate algorithm that generates the tANS tables using only the known frequencies of symbols without processing the sequence itself (however, methods that analyze the sequence might avoid this effect \([12], [15]\); these methods are usually infeasible in practice due to their relatively slow performance).

The second contribution of the present paper is a modification of the range ANS (rANS). The rANS is another variant of ANS invented by Duda. It has a number of advantages in some use scenarios because of which it was favoured by some practitioners (and because it is easier to learn). The rANS was first noticeably slower in practice than the tANS but its current SIMD implementations tend to be at least on a par with tANS. The main advantage of rANS is that it does not need tANS tables (however, fast variants still require some specific tables). Due to the less wasteful use of memory, the rANS is more suitable for (pseudo)adaptive compression or when several streams of data are encoded simultaneously \([7], [18], [19], [31]\). An inherent problem of rANS that slows down it significantly is a necessity to perform the integer division during the encoding, which is expensive on modern processors. The proposed modification of rANS, which we call rANS with fixed accuracy, tries to mitigate this issue making the encoder faster while preserving the same good properties of rANS.

The new rANS takes as a regulated parameter an integer \(k \geq 1\). It is guaranteed that the division can be performed only in cases when its result belongs to the range \([2^k, 2^{k+1})\). Thus, the division can be computed by faster methods provided \(k\) is small. We upperbound the redundancy for the rANS with fixed accuracy \(k\) by \(\frac{n}{2^k} \log e + r\). The new rANS variant is faster than the standard rANS but is still not as fast as tANS: in our experiments, the rANS with fixed accuracy \(k = 3\) was two times slower than tANS implemented by Collet \([9]\) (however, we did not use the technique of interleaving streams by Giesen \([19]\) in our implementation). We believe that the rANS with fixed accuracy might be more suitable for hardware implementations, where its restricted division operation can be sped up more efficiently using versatile capabilities for parallelization.

The paper is organized as follows. After short preliminaries, Section III introduces a simple variant of the range ANS and, based on it, the tabled ANS. Section IV presents a tight analysis of the redundancy for the tabled ANS, upper and lower bounds. Section V describes a novel variant of range ANS and analyzes it. We conclude with some open problems in Section VI.

## II. Preliminaries

A symbol is an element of a finite set of integers called an alphabet. We consider sequences of symbols. The sequences are also sometimes called strings. Integer intervals are denoted by \([i..j] = \{k \in \mathbb{Z}: i \leq k \leq j\}\) and \([i..j) = [i..j] \setminus \{j\}\). Fix an alphabet \([0..\sigma)\) with \(\sigma\) symbols and a sequence of symbols \(a_1, a_2, \ldots, a_n\) from it. For each symbol \(a\), denote by \(f_a\) the number of occurrences of \(a\) in the sequence. The number \(f_a\) is called the frequency of \(a\). The number \(f_a/n\) is the empirical probability of \(a\). An entropy encoder transforms this sequence into a sequence of bits that is then transmitted to a decoder. The encoder usually also transmits to the decoder the table of frequencies \(f_a\). However, the problem of storage for the table is out of the scope of the present paper and we focus only on the encoding for the sequence itself implicitly assuming that both sides know the length \(n\) of the sequence and the frequencies of symbols. The following quantity is used as an information theoretic lower bound \([11], [21]\) for the number of bits that any encoder should spend for the sequence encoding in the worst case (even under the assumption that the table of frequencies \(f_a\) is known):

\[
\sum_{a \in [0..\sigma)} f_a \log \frac{n}{f_a},
\]

assuming that \(f_a \log \frac{n}{f_a} = 0\) whenever \(f_a = 0\). We call \((1)\) the entropy formula (though the empirically calculated entropy itself, i.e., the empirical entropy, is defined as the quantity \((1)\) divided by \(n\)). The difference between the number of bits produced by an encoder and the optimal number of bits from \((1)\) is called a redundancy. The redundancy is the primary focus of our analysis of ANS.

For each symbol \(a\), denote \(c_a = \sum_{a' \in [0..a)} f_{a'}\). Typically, encoders store two arrays of \(\sigma\) numbers \(c_a\) and \(f_a\) and some additional tables necessary for encoding. The ANS encoders are not an exception.

**Remark 1:** We digress at this point to discuss how the quantity \((1)\) can be interpreted as a lower bound. The quantity \((1)\) is equal to \(\min \log(1/ Pr(Q produces \(a_1, a_2, \ldots, a_n\)))\), where the minimum is taken for all memoryless sources \(Q\) randomly generating sequences of length \(n\) over the alphabet \([0..\sigma)\) (see \([17]\)). If one restricts attention only to encoders that produce uniquely decodable codes (for example, prefix-free codes), then \((1)\) is a lower bound for the expected number of bits produced by such encoder provided each symbol of the length-\(n\) input sequence is generated by the memoryless source with respective probabilities \(f_a/n\) of the symbols \([11], [24]\). It is this fundamental fact that seems to justify the usage of \((1)\) as a lower bound in \([21], [22]\) and numerous subsequent papers.

However, as it was mentioned in the introduction, we aim to avoid any probabilistic assumptions on the source in the input. In this case, the tight information theoretic lower bound for the length of the uniquely decodable code is \(\lceil \log \frac{n!}{f_0!f_1! \cdots f_{\sigma-1}!} \rceil\), provided the length \(n\) of the sequence and the table of frequencies \(f_a\) are accessible to the encoder and decoder for free. This quantity is less than \((1)\) but close: the difference is at most \(O(\sigma \log \frac{n}{\sigma})\). If the table of frequencies cannot be accessed by the
decoder for free and, thus, should be transmitted somehow, then the lower bound for the length of the uniquely decodable code is $(1)\ plus \ \Theta(\sigma \log \frac{2}{\sigma})$ when $\sigma < n$ (see [27], [30], [32]). Therefore, $(1)$ is indeed a lower bound in this case but it is not tight (recall that this is not our setting: we consider the frequencies as freely transmitted to the decoder).

It is still convenient to refer to $(1)$ as a kind of "optimal lower bound" and, following the established tradition, we call it as such in the sequel, despite the fact that it is not exactly the case.

III. ANS ENCODERS

On a high level, an ANS encoder can be described as a data structure that maintains a positive integer $w$ that can be modified by two stack operations “$w = \text{push}(w, a)$” and “$(w, a) = \text{pop}(w)$”: push encodes a symbol $a$ into the number $w$ and returns the modified value for $w$, and pop performs a reverse operation decoding the symbol $a$ and restoring the old value for $w$. Given a sequence of symbols $a_1, a_2, \ldots, a_n$, the encoder pushes them consecutively and transmits the resulting integer $w$ to a decoder, which in turn retrieves the sequence from $w$ performing pop operations. Note that the symbols appear in the reverse order during the decoding, which is a distinctive feature of ANS. For their correct coordinated work, both encoder and decoder should receive in advance the same table of (approximate) frequencies of symbols in the sequence $a_1, a_2, \ldots, a_n$. However, in what follows, we significantly diverge from this standard way of explanation for ANS and introduce only the streaming ANS but in an unconventional manner; the unbounded ANS variant is not discussed at all.

A. Basic Range ANS

Suppose that we are to encode a sequence of symbols $a_1, a_2, \ldots, a_n$ from the alphabet $\{0, \ldots, \sigma\}$. For each symbol $a \in \{0, \ldots, \sigma\}$, denote by $f_a$ its frequency in the sequence (i.e., its empirical probability is $\frac{f_a}{n}$). We assume that $n$ is a power of two, i.e., $n = 2^r$ for an integer $r \geq 0$. It is a standard assumption for both ANS and arithmetic coding that simplifies implementations. If the length of the sequence is not a power of two, then either the real probabilities of symbols are approximated with numbers as such in the sequel, despite the fact that it is not exacty the case.

On a high level, an ANS encoder can be described as a data structure that maintains a positive integer $w$ and integers $i, r$.

The streaming ANS encoder reads the symbols $a_1, a_2, \ldots, a_n$ from left to right and, after processing $a_1, a_2, \ldots, a_i$, encodes the processed part into a positive integer $w_i$. Initially, $w_0 = 2^r$; the choice for $w_0$ is somewhat arbitrary, the only necessary condition is $w_0 \geq 2^r$. To encode a new symbol $a_{i+1}$, we transform the number $w_i$ into $w_{i+1}$ by increasing the value stored in the $r+1$ highest bits of $w_i$. Thus, $w_0 < w_1 < \cdots < w_{i+1}$. According to our terminology, we have $w_{i+1} = \text{push}(w_i, a_{i+1})$.

It is instructive to image the number $w_i$ as a bit stream written from the highest bit (which is always 1) to lowest bits. We replace at most $r+1$ highest bits with a new larger value; see Figure 1.

$$w_i = \frac{10010011000100110110}{1010011101100100010110}$$

$$w_{i+1} = \frac{10010011000100110110}{r+1 \text{ bits}}$$

Fig. 1. A transformation of the number $w_i$ into $w_{i+1}$.

The numbers $w_i$ might be very long. However, since only the highest $r+1$ bits of $w_i$ matter for the encoder, all lower bits can be stored in an output buffer. The integer $r$ is chosen so that the $r+1$ bits can be stored into one machine word. For simplicity of the execution, we omit this technical detail and continue to discuss the numbers $w_i$ as if they were stored explicitly.

The goal is to encode the whole sequence optimally so that, ideally, the final number $w_n$ occupies $\sum_a f_a \log \frac{n}{f_a}$ bits. Intuitively, one can achieve this by encoding each symbol $a$ into $\log \frac{n}{f_a} = r - \log f_a$ bits. During the processing of $a = a_{i+1}$, we could have achieved this by replacing the highest $\log f_a + 1$ bits of $w_i$ with $r+1$ bits that store a new larger value; then, the number of bits in $w_{i+1}$ and $w_i$ will differ by $r+1 - (\log f_a + 1) = \log \frac{n}{f_a}$. But the number $\log f_a$ is not integer in general. Therefore, instead, we replace either $\lfloor \log f_a \rfloor + 1$ or $\lceil \log f_a \rceil + 2$ highest bits of $w_i$ with new $r+1$ bits. As a result, the number of bits in $w_{i+1}$ increases by either $r - \lfloor \log f_a \rfloor$ or $r - \lceil \log f_a \rceil - 1$, which is approximately $\log \frac{n}{f_a}$. The cumulative growth of $w_n$ may approach the optimal $\log \frac{n}{f_a}$ bits per symbol $a$ on average if the case when $\lfloor \log f_a \rfloor + 2$ bits are replaced happens more often. (Note that when $f_a \geq n/2 = 2^{r-1}$, we have $\lfloor \log f_a \rfloor = r - 1$ and the number of bits in $w_i$ sometimes might not change at all; but the content will change.)

What is the content of the $r+1$ new highest bits in $w_{i+1}$ and how do we decide whether $\lfloor \log f_a \rfloor + 1$ or $\lceil \log f_a \rceil + 2$ highest bits of $w_i$ will be replaced?

Denote by $x'$ the value stored in the highest $r+1$ bits of $w_{i+1}$, i.e., $w_{i+1} = x' \cdot 2^{\lfloor \log w_{i+1} \rfloor - r + \Delta}$, where $0 \leq \Delta < 2^{\lfloor \log w_{i+1} \rfloor - r}$ and $2^r \leq x' < 2^{r+1}$. Denote by $x$ the value of the highest bits of $w_i$ that were replaced with $x'$, i.e., $w_i = x \cdot 2^{\lfloor \log w_{i+1} \rfloor - r + \Delta}$,
where \( x \geq 1 \). (In the example of Figure 1, \( x \) and \( x' \) are emphasized and \( \Delta \) is a common part of \( w_i \) and \( w_{i+1} \).) The scheme must be reversible: the number \( x' \) must provide sufficient information for the decoder to restore the symbol \( a_{i+1} \) and the number \( x \). Since \( \sigma \leq \sum a f_a = 2^r \), there is enough place to encode \( a_{i+1} \) into the \( r \) lowest bits of \( x' \) (the highest, \( (r+1) \)bit of \( x' \) is 1). “Excessive” \( 2^r - \sigma \) possible values of \( x' \) will be used to restore the number \( x \), the replaced highest bits of \( w_i \).

The \( r \) lowest bits of \( x' \) can store any number from the range \([0..2^r] \). The encoder chooses one of these numbers depending on the values of \( x \) and \( a_{i+1} \). We distribute the numbers from \([0..2^r] \) among the symbols according to their frequencies: for each symbol \( a \), denote \( c_a = \sum_{a \in c_a} f_a \); the subrange of values \([c_a..c_{a+1}] \) is allocated for \( a \). Hence, given a symbol \( a = a_{i+1} \), we have a room in the range that can store any number from \([0..f_a] \) and this information should suffice to restore the replaced number \( x \) from \( x' \). As an example, Figure 2 depicts a distribution of the range \( 2^r = 2^4 \) among symbols \( a, b, c, d \) with frequencies \( 3, 5, 6, 2, \) respectively.

\[
\begin{array}{cccccccccc}
0 & a & a & a & b & b & b & c & c & c & d & d & 2^r \\
\end{array}
\]

\textbf{Fig. 2.} A distribution of symbols in a range of length \( 2^r \).

Let \( a = a_{i+1} \). Denote by \( x_1 \) and \( x_2 \) the values stored in, respectively, the highest \([ \log f_a ] + 1 \) and the highest \([ \log f_a ] + 2 \) bits of \( w_i \). We assume that \( x = x_1 \) if \( x_1 \geq f_a \), and \( x = x_2 \) otherwise. Thus, the condition \( x_1 \geq f_a \) determines whether \([ \log f_a ] + 1 \) or \([ \log f_a ] + 2 \) highest bits of \( w_i \) will be used for \( x \). Note that \( 2^{[\log f_a]} \leq x_1 < 2 \cdot f_a \) and \( x_2 < 2 \cdot x_1 + 1 \). Therefore, we have \( f_a \leq x < 2 \cdot f_a \) and, thus, \( x \mod f_a = x - f_a \). The latter can be clearly seen in Figure 3: the bits occupied by \( x_1 \) and \( x_2 \) are emphasized, respectively, on the left and right; note that the number \( f_a \) occupies \([ \log f_a ] + 1 \) bits.

\[
\begin{array}{cc}
\begin{array}{l}
w_i = 1011001100100010110 \\
f_a = 10011 \\
x \mod f_a = 11
\end{array}
\end{array}
\]

\textbf{Fig. 3.} Two cases: \( x \) occupies \([ \log f_a ] + 1 \) (left) or \([ \log f_a ] + 2 \) (right) highest bits of \( w_i \).

In order to restore the value \( x \) from \( x' \), it suffices to encode somehow the value \( x \mod f_a \) into \( x' \); then, \( x \) is restored as \( x = f_a + (x \mod f_a) \). To this end, the range \([c_a..c_{a+1}] \) allocated for the symbol \( a \) has exactly enough room. Thus, we have the following transformation to encode \( a \):

\[
x' = 2^r + c_a + (x \mod f_a), \quad \text{where} \quad f_a \leq x < 2 \cdot f_a. \tag{2}
\]

With this approach, the decoder should perform a reverse transformation for the \( r + 1 \) highest bits of the number \( w_{i+1} \) in order to restore \( w_i \). The decoding is straightforward:

\[
x = f_a + (x' \mod 2^r) - c_a, \quad \text{where} \quad c_a \leq x' \mod 2^r < c_{a+1}. \tag{3}
\]

The decoded symbol \( a \) is determined by examining to which range \([c_a..c_{a+1}] \) the number \( x' \mod 2^r \) belongs. This is how the operation \( \langle w_i, a_{i+1} \rangle = \text{pop}(w_{i+1})^r \) is performed. It remains to observe that \( x' > x \) since \( x' \geq 2^r + (x \mod f_a) > f_a + (x \mod f_a) = x \). Therefore, the number \( w_{i+1} \) indeed is larger than \( w_i \).

The described scheme is the simplest form of the so-called range ANS (rANS). As will be seen later, the size of \( w_n \) in bits can be bounded by \( \sum_a f_a \log f_a + O(n) \). The redundancy \( O(n) \) is quite significant for many applications and, indeed, the described scheme does not perform well in practice. The following section describes an additional “shuffling” step in the encoder that fixes this issue. There exists, however, a more elaborate version of the range ANS that does not have such a problem and works well without shuffling; we discuss it briefly in Section where we also present a novel variant of the range ANS with good compression guarantees.

\section{B. Shuffling and Tabled ANS}

The idea of the shuffling step enhancing the simple range ANS is to shuffle the lower bits of \( x' \) in a random-like fashion. Thereby, the scheme (2) is changed as follows:

\[
x' = 2^r + \text{shuffle}[c_a + (x \mod f_a)], \quad \text{where} \quad f_a \leq x < 2 \cdot f_a. \tag{4}
\]

The array \( \text{shuffle}[0..2^r - 1] \) is a permutation of the range \([0..2^r] \) but it is not entirely random: in order to guarantee the inequality \( x' > x \) (which implies \( w_{i+1} > w_i \)), it must satisfy the following property:

\[
\text{shuffle}[c_a + i] < \text{shuffle}[c_a + j], \quad \text{whenever} \quad 0 \leq i < j < f_a. \tag{5}
\]
Due to this condition, we have \( x' > x \) since \( x' \geq 2^r + (x \mod f_a) > f_a + (x \mod f_a) = x \). It is convenient to view shuffle as defined via an additional array \( \text{range}[0..2^r-1] \) that stores an (arbitrary) permutation of the array of symbols from Figure 2 in which every symbol \( a \) occurs exactly \( f_a \) times; then, for \( i \in [0..f_a) \), \( \text{shuffle}(c_a + i) \) is equal to the index of the \((i+1)\)th occurrence of the symbol \( a \) in range (see Figure ??). Thus, to define shuffle, it suffices to initialize the array range; we will implicitly imply this relation in the sequel when the initialization of shuffle is discussed.

The decoding procedure (3) performs a reverse transformation in an obvious way:

\[
x = f_a + \text{unshuffle}[x' \mod 2^r] - c_a, \quad \text{where range}[x' \mod 2^r] = a.
\]

Note the use of the array range in the decoder to determine the symbol \( a \). The array unshuffle is the inverse of shuffle such that \( \text{unshuffle}[	ext{shuffle}[z]] = z \), for any \( z \in [0..2^r) \). Thus, it “moves” the value \( x' \mod 2^r \) to its “correct” location and we have to add \( f_a \) and subtract \( c_a \) afterward. However, implementations usually construct, instead of the arrays unshuffle and range, an array decode that stores, for each number \( x' \mod 2^r \), an already corrected value for \( x \) and the corresponding symbol \( a \); hence, the decoding is much simpler:

\[
(x, a) = \text{decode}[x' \mod 2^r].
\]

The described scheme is called a tabled ANS (tANS). It is the most popular variant of ANS widely used in practice. The choice of the shuffling method is crucial for its performance. Some methods are considered in the next section. There are several additional technical improvements that can be applied to this basic scheme. Perhaps, the most notable of them is that one can feed to the decoder more than \( r+1 \) bits at once, decoding many symbols in one step (the information about the decoded symbols and the new value for \( x \) must be stored in the array decode). Also, we point out again that \( x \mod f_a \) in (4) is computed as \( x - f_a \) since \( f_a \leq x < 2 \cdot f_a \). We do not discuss these details further.

### C. Shuffling Methods

The general rule for shuffling is to distribute symbols in the array range as uniformly as possible so that, for any symbol \( a \), the distance between consecutive occurrences of \( a \) is approximately \( \frac{n}{f_a} \). Implementations usually use heuristics for this or Duda’s method [14] (which is introduced below). Let us discuss some considerations on this regard that will be developed in a more rigorous way in Section IV.

The following informal argument justifies the scheme with shuffling. The value \( \log x' - \log x \) is, in a sense, an increase in bits from the number \( w_i \) to \( w_{i+1} \); the bit length of \( w_n \), the final encoding, is approximately the sum of the increases (this reasoning is formalized in Section IV). Denote \( \delta = x \mod f_a \). We have \( x = f_a + \delta \). If the symbols in the array range are distributed uniformly, the distance between two consecutive symbols \( a \) is approximately \( \frac{n}{f_a} \). Therefore, using (4), the encoder transforms \( x \) approximately to \( x' \approx 2^r + \frac{n}{f_a} \delta = \frac{n}{f_a} (f_a + \delta) = \frac{n}{f_a} x \) (recall that \( n = 2^r \)). Hence, we obtain \( \log x' - \log x \approx \log \frac{n}{f_a} \), which is precisely the optimal number of bits for the symbol \( a \) according to the entropy formula.

The argument suggests that the shuffling method should spread the symbols in the array range in such a way that the distance between two consecutive occurrences of \( a \) is approximately \( \frac{n}{f_a} \) and the encoder transforms the number \( x = f_a + \delta \) as close as possible to the number \( 2^r + \frac{n}{f_a} \delta \). Under this assumption of “uniformity”, if the first occurrence of symbol \( a \) is at position \( p \) in range, then \( x \) is transformed into \( x' \approx 2^r + \frac{n}{f_a} \delta + p \). The term \( p \) adds to the redundancy associated with \( f_a \) symbols \( a \): the larger the value of \( p \), the more bits are spent per symbol \( a \). Therefore, the first occurrences of more frequent symbols should be closer to the beginning of the array range so that they produce less redundancies overall. Duda’s method, which he called a precise initialization, tries to take into account all these considerations.

Duda’s algorithm maintains a priority queue with the following operations: \( \text{put}(q, a) \) adds a pair of numbers \((q, a)\) to the queue; \((q, a) = \text{getmin}()\) removes from the queue a pair \((q, a)\) with the smallest value \( q \) (breaking ties arbitrarily). We first give in Algorithm 1 a simplified Duda’s algorithm, which is easier to analyze.

**Algorithm 1** A simple initialization algorithm.

```python
for a ∈ [0..σ) do
    put(0, a), d_a = c_a;
for i = 0, 1, . . . , 2^r − 1 do
    (q, a) = getmin(), range[i] = a, shuffle[d_a] = i;
    put(q + n/ f_a, a), d_a = d_a + 1;
```

The array range corresponding to shuffle is not needed for the encoder and its construction is added here for the convenience of the reader.

At every moment during the work of the algorithm, there is only one instance of each symbol in the priority queue and, if a symbol \( a \) was assigned to the array range exactly \( k \) times, then it is represented by the pair \((n, k, a)\) in the queue. Therefore, after \( f_a \) assignments of \( a \) into range, the symbol is represented by \((n, a)\) and all other symbols \( b \) that had less than
\(f_a\) assignments in range are represented as \((q,b)\) with \(q < n\). Hence, in the end, each symbol \(a\) occurs in range exactly \(f_a\) times.

Duda’s original “precise initialization” is the same as Algorithm [1] except that the operation “put(0, \(a\))” from the first loop is changed to “put(\(\frac{1}{2} \cdot \frac{n}{f_a} , a\))”. Its correctness is proved analogously.

IV. Analysis of the Tabled ANS

We are to estimate the redundancy of the output produced by the ANS encoder, i.e., the difference between \(\lceil \log w_n \rceil\) and the lower bound \(\sum_a f_a \log \frac{n}{f_a}\). We postpone the analysis of the ANS without shuffling to the next section, where its more general variant is considered. In this subsection we consider the ANS with shuffling, i.e., the tabled ANS (tANS). To the best of our knowledge, the following analysis, albeit quite simple, evaded the attention of researchers and was not present in prior works. Our proof methods, however, stem from observations and arguments from Dubé and Yokoo [37] and Duda [13], [14].

A. Upper Bounds

Let us upperbound \(w_{i+1} - w_i = \log \frac{w_{i+1}}{w_i}\), a bit increase after one step of the encoding procedure. The total number of bits will be then estimated as follows (the term \(r\) appears because \(w_0 = 2^r\) and \(\log w_0 = r\)):

\[
\log w_n = \log \frac{w_n}{w_{n-1}} + \log \frac{w_{n-1}}{w_{n-2}} + \cdots + \log \frac{w_1}{w_0} + r. \tag{6}
\]

Suppose that \(w_{i+1}\) was obtained from \(w_i\) by “inserting” a symbol \(a\) as described above. Denote \(\ell = \lfloor \log w_{i+1}\rfloor - r\) so that \(w_i = x \cdot 2^\ell + \Delta\) and \(w_{i+1} = x' \cdot 2^\ell + \Delta\), where \(0 \leq \Delta < 2^\ell\). Then, \(\log \frac{w_{i+1}}{w_i} = \log \left(\frac{x'2^\ell + \Delta}{x2^\ell + \Delta}\right) = \log x' - \log x + \log \left(\frac{1+\Delta/(x'2^\ell)}{1+\Delta/(x2^\ell)}\right)\). Since \(x' > x\), the additive term \(\log \left(\frac{1+\Delta/(x'2^\ell)}{1+\Delta/(x2^\ell)}\right)\) is negative and, thus, we have obtained the following inequality:

\[
\log w_{i+1} - \log w_i \leq \log x' - \log x. \tag{7}
\]

It remains to estimate how close is \(\log x' - \log x\) to the optimum \(\frac{\sigma}{f_a}\). We first consider the case when the encoder uses the shuffling produced by simplified Duda’s algorithm (Algorithm [1]).

Fix a symbol \(a\) and a number \(\delta \in [0..f_a]\). Denote by \(k\) the index of the \((\delta+1)\)th occurrence of \(a\) in range. Note that shuffle[c_a + \delta] = k, by definition. For each \(b \in [0..\sigma]\), denote by \(k_b\) the number of symbols \(b\) in the subrange range[0..k-1]. Clearly, we have \(k = \sum_b k_b\). The shuffling algorithm implies the following inequality:

\[
(k_b - 1) \frac{n}{f_b} \leq \delta \frac{n}{f_a}. \tag{8}
\]

We express \(k_b\) from (8) as \(k_b \leq \delta \frac{f_b}{f_a} + 1\). Summing over all \(b \in [0..\sigma]\), we deduce from this \(k \leq \delta \frac{n}{f_a} + \sigma\). It follows from (7) that, in order to analyze the number of bits per symbol produced by the encoder, we have to estimate \(\log x' - \log x\), where, by (4), \(x' = 2^\ell + \text{shuffle}[c_a + (x \mod f_a)]\). Assuming \(\delta = x \mod f_a\), we obtain \(x = f_a + \delta\) and \(x' = 2^\ell + k = n + k\). Therefore, \(\log x' - \log x = \log (n+k) - \log (n+\delta \frac{f_a}{f_a} + \sigma) - \log x = \log \left(\frac{f_a}{f_a} + 1 + \delta \frac{f_a}{f_a} + \sigma\right) - \log x = \log \frac{f_a}{f_a} + \log (1 + \delta \frac{f_a}{f_a} + \sigma) \leq \log \frac{f_a}{f_a} + \log e\). Thus, we estimate the number of bits per symbol \(a\) as \(\log \frac{f_a}{f_a} + \frac{\sigma}{f_a}\) e, i.e., the redundancy is \(\frac{\sigma}{f_a}\) log e bits per symbols.

Now let us analyze Duda’s original algorithm. The algorithm is the same as Algorithm [1] except that the operation “put(0, \(a\))” from the first loop is changed to “put(\(\frac{1}{2} \cdot \frac{n}{f_a} , a\))”. The analysis is slightly more complicated. First, an equation analogous to (3) for this case looks as follows:

\[
(k_b - 1) \frac{n}{f_b} \leq \left(\delta + \frac{1}{2}\right) \frac{n}{f_a}. \tag{9}
\]

We then similarly deduce \(k_b \leq (\delta + \frac{1}{2}) \frac{f_b}{f_a} + \frac{1}{2}\) and, summing over all \(b\), \(k \leq (\delta + \frac{1}{2}) \frac{n}{f_a} + \frac{n}{f_a}\). Again, assuming \(x = f_a + \delta\), it follows from this that \(\log x' - \log x \leq \log (\frac{f_a}{f_a} + \frac{n}{f_a} + \frac{1}{2}) - \log x = \log \frac{n}{f_a} + \log (1 + \frac{1}{2} + \frac{\sigma}{f_a}) \leq \log \frac{n}{f_a} + (\frac{1}{2} + \frac{\sigma}{f_a}) \log e\). The symbol \(a\) occurs in the sequence exactly \(f_a\) times. Hence, the redundancies \(\frac{1}{2} \log e\) for symbols \(a\) sum up to \(\frac{1}{2} \log e\) over all these occurrence and, therefore, in the end we obtain \(\frac{\sigma}{f_a} \log e\) bits per symbol contributed by the terms \(\frac{1}{2} \log e\), for all symbols \(a\), in the final encoding. Adding to this the \(\frac{\sigma}{f_a} \log e\) bits per symbol, we obtain \(\frac{n}{f_a} \log e\) redundant bits per symbol in the final encoding.

It remains to add the additive term \(r\) from (6) to the redundancy, which contributes \(\frac{r}{n}\) bits per symbol, and the following theorem is proved.

**Theorem 1:** Given a sequence \(a_1, a_2, \ldots, a_n\) of symbols from an alphabet \([0..\sigma]\) such that each symbol \(a\) occurs in it \(f_a\) times and \(n = 2^r\) for an integer \(r\), the ANS encoder using simplified Duda’s precise initialization transforms this sequence into a bit string of length

\[
\sum_{a \in [0..\sigma]} f_a \cdot \log \frac{n}{f_a} + O(\sigma + r),
\]

where the \(O(\sigma + r)\) term can be bounded by \(\sigma \log e + r\). Thus, we have \(O(\frac{\sigma + \sigma}{n})\) redundant bits per symbol.
In practice, the length $m$ of the encoded sequence $a_1, a_2, \ldots, a_m$ is not necessarily a power of two. A typical solution for this case is to approximate the real empirical probabilities $f_a/m$ of symbols with approximate ones $\hat{f}_a/2^r$, where $\sum_a \hat{f}_a = 2^r$. The ANS encoder then processes the sequence as usually but using the “frequencies” $\hat{f}_a$ instead of $f_a$. The same analysis can be applied for this case: Equations (6) and (7) trivially hold and the value $\log x' - \log x$ is bounded in the same manner by $\log \frac{2^r + \hat{f}_a}{\hat{f}_a} \log e$, for simplified Duda’s initialization, and by $\log \frac{2^r}{\hat{f}_a} + (\frac{1}{2^r} + \frac{\hat{f}_a}{2^r}) \log e$, for Duda’s initialization. Summing the redundancies over all $m$ symbols, we obtain the following theorem (for simplicity, the theorem is stated only for simplified Duda’s initialization).

**Theorem 2**: Let $a_1, a_2, \ldots, a_m$ be a sequence of symbols from an alphabet $\{0, \sigma\}$ such that each symbol $a$ occurs in it $f_a$ times. Let the probabilities $f_a/m$ be approximated by numbers $f_a/n$ such that $f_a$ are integers, $n = 2^r$, for an integer $r$, and $\sum_a f_a = n$. The ANS encoder that uses the approximate probabilities and simplified Duda’s initialization transforms this sequence into a bit string of length

$$\sum_{a \in \{0, \sigma\}} f_a \cdot \log \frac{n}{f_a} + O\left(\frac{\sigma m}{n} + r\right),$$

where the $O\left(\frac{\sigma m}{n} + r\right)$ term can be bounded by $\frac{\sigma m}{n} \log e + r$.

**B. Lower Bound Example**

Apparently, the $r$-bit redundancy incurred by the initial value $w_0$ is unavoidable in the described scheme. It is less clear whether an $O(\sigma)$ additive term is necessary in Theorem 1. An informal argument supporting that this is the case is as follows. Consider a sequence in which all symbols are (approximately) equiprobable, i.e., their frequencies $f_a$ are $\sim \frac{1}{\sigma}$. The lower bound for the encoding of the sequence is $n \log \sigma$ bits. The array range constructed by Duda’s initialization algorithm for the sequence looks (approximately) as $n/\sigma$ blocks, each of which is of size $\sigma$ and consists of consecutive symbols $0, 1, \ldots, \sigma - 1$. Hence, when the encoder receives a symbol $a_{i+1} = a \in \{0, \sigma\}$ during its work, it transforms the number $x = f_a + \delta$, where $\delta = x \mod f_a$, occupying leading bits of $w_i$, into the number $x' = 2^r \frac{n}{f_a} \delta + a = \frac{n}{f_a} x + a = \sigma x + a$ (note that $f_a = \sigma / n$, by our assumption). As in the previous section, one can deduce from this that $\log x' - \log x = \log \sigma + \log(1 + \frac{a}{\sigma})$. Since $x < 2^r f_a = 2^r n/\sigma < 1$, the redundant additive term $\log(1 + \frac{a}{\sigma})$ can be estimated as $\log(1 + \frac{a}{\sigma}) < \log(1 + \frac{1}{n})$; we used the inequality $\log(1 + z) \geq 0$, where $0 \leq z < 1$. Thus, the redundancy is approximately $\frac{\sigma}{n}$ bits per symbol $a$, which sums up to the total redundancy of $\sum_a \frac{\sigma}{n} f_a = \sum_a \frac{\sigma}{n} = \frac{\sigma}{n} - 1$ bits over all symbols in the sequence.

This informal argument is only an intuition since the negative terms $\log \frac{1 + \frac{a}{\sigma}}{1 + \frac{a}{\sigma}}$ that appear in the analysis of Section IV-A could, in principle, diminish the described effect. Nevertheless, as we are to show, an $\Omega(\sigma)$ redundancy indeed appears in some instances.

Fix an even integer $r > 0$. Observe that $2^r \equiv 1 \pmod{3}$ since $r$ is even. Denote $n = 2^r$. The sequence under construction will contain $\sigma = (n - 1)/3 + 1$ symbols $0, 1, \ldots, \sigma - 1$. Each symbol $a \in \{0, \sigma - 1\}$ has exactly three occurrences in the sequence (i.e., $f_a = 3$) and the symbol $\sigma - 1$ occurs only once (i.e., $f_{\sigma - 1} = 1$); note that $\sum a \in \{0, \sigma\} f_a = 3(\sigma - 1) + 1 = n$. The entropy formula gives the following lower bound on the encoding size for the sequence:

$$(n - 1) \log \frac{n}{3} + \log n = (n - 1)(r - \log 3) + r = (n - 1)(r - 1.58496\ldots) + r.$$  \hfill (9)

It is straightforward that with such frequencies of symbols both Duda’s initialization algorithm and its simplified variant (Algorithm 1) construct the same array range: the subrange range$[0, \sigma - 1]$ contains consecutively the symbols $0, 1, \ldots, \sigma - 1$ (in this order) and the subranges range$[\sigma - 2, \sigma - 2]$ and range$[2\sigma - 1, n - 1]$ are equal and both contain consecutively the symbols $0, 1, \ldots, \sigma - 2$ (in this order). Now let us arrange the symbols in the sequence $a_1, a_2, \ldots, a_n$.

The last symbol $a_n = \sigma - 1$. The rest, $a_1, a_2, \ldots, a_{n-1}$, consists of symbols $a \in \{0, \sigma - 1\}$ whose frequencies are $f_a = 3$ (112 in binary). When the encoder processes a symbol $a_{i+1} = a \in \{0, \sigma - 1\}$ and modifies the number $w_i$ representing the prefix $a_1, a_2, \ldots, a_i$, it replaces either two or three leading bits of $w_i$ with new $r + 1$ bits, thus producing the number $w_{i+1}$. The choice of whether to replace two or three bits depends on whether the two leading bits of $w_i$ are 11 or 10, respectively (i.e., whether the two bits store a number less than $f_a = 3$ or not). We are to arrange the symbols $0, \ldots, \sigma - 1$ in the sequence $a_1, a_2, \ldots, a_{n-1}$ in such a way that the encoder chooses the two options alternatingly: it replaces three leading bits of $w_i$ if $i$ is even, and two bits if $i$ is odd ($i = 0, 1, \ldots, n - 2$). The total number of bits produced in this way is at least $\frac{n}{3} (r - 2) + \left(\frac{\sigma}{3} - 1\right)(r - 1) + r$ (the additive term $r$ appears when the last symbol $a_n = \sigma - 1$ is encoded), which is equal to $(n - 1)(r - 1.5) + r - 0.5$. Comparing this to (9), one can see that the encoding generated by ANS is larger than the optimum (9) by at least $(\log 3 - 1.5)(n - 1) - 0.5 > 0.08496(n - 1)$ (the estimation holds for large enough $n$, so that the term 0.5 disappears). By simple calculations, we deduce from the equality $\sigma = (n - 1)/3 + 1$ that the redundancy $0.08496(n - 1)$ is larger than $\frac{\sigma}{n}$. Let us describe an arrangement of symbols that produces such effect of “alternation”.

The encoder consecutively transforms the initial value $w_0 = 2^r$ into $w_1, w_2, \ldots$ by performing the push operations: $w_{i+1} = \text{push}(w_i, a_{i+1})$. Let us call the number $\lfloor w_i/2^{\frac{\log w_i}{r}} - r \rfloor \mod 2^r$ a **state**; it is the value stored in the highest $r + 1$ bits of the number $w_i$ currently processed by the encoder minus the highest bit 1. The range of possible states is $[0..2^r - 1]$. We split
The states thus “bounce” between the segments A and C during the processing of $a_0$ from the segment A, any state from the segment C transits to a state from the segment A when the encoder receives a symbol $a_1$. This behaviour is determined by first bits of $w_i$, which are equal to 11... only for states from the segment C; see Figure 5. The execution of the encoder starts with the state 0, which belongs to the segment A. We are to arrange symbols in the sequence $a_1, a_2, \ldots, a_{n-1}$ so that, for odd $i$, the state corresponding to $w_i$ belongs to the segment C, and for even $i$, to the segment A. The states in our arrangement will never belong to the segment B.

Since $\sigma + \sigma/2 - 1 = n/2$ and $n/2$ is the leftmost state from the segment C, any state from the segment A transits to a state from the segment C when the encoder receives a symbol $a$ from the range $[\sigma/2 - 1..\sigma - 1]$; the new state is $\sigma + a$, the $a$th element of the segment ii (see Figure 5 for an illustration). Similarly, since $\sigma/2 < n/4$ and $n/4 - 1$ is the rightmost state from the segment A, any state from the segment C transits to a state from the segment A when the encoder receives a symbol $a$ from $[0..\sigma/2)$. Accordingly, for $i \in [0..n-1)$, we put in the sequence as the symbol $a_{i+1}$ a symbol from $[\sigma/2 - 1..\sigma - 1)$ if $i$ is even, and a symbol from $[0..\sigma/2)$ if $i$ is odd (note that both ranges share a common symbol $\sigma/2 - 1$; it is not a mistake).

The states thus “bounce” between the segments A and C during the processing of $a_1, a_2, \ldots, a_{n-1}$ by the encoder. Since the sizes of both ranges $[0..\sigma/2)$ and $[\sigma/2 - 1..\sigma - 1]$ are $\sigma/2$ and they share a common symbol $\sigma/2 - 1$, the symbols $[0..\sigma-1)$ can be distributed in the sequence $a_1, a_2, \ldots, a_{n-1}$ in such way that each symbol occurs exactly three times and each symbol $a_{i+1}$ is from the range $[\sigma/2 - 1..\sigma - 1)$, for even $i$, and from the range $[0..\sigma/2)$, for odd $i$.

We thus have obtained a redundancy of $\frac{\sigma - 1}{4}$ bits. Adding to this $r - 2$ bits produced by the $r - 2$ lowest bits of the initial value $w_0$, we have proved the following theorem.

**Theorem 3:** For arbitrarily large $n = 2^r$, there exists a sequence $a_1, a_2, \ldots, a_n$ of symbols from an alphabet $[0..\sigma)$ with $\sigma > n/3$ such that the ANS encoder using [simplified] Duda’s precise initialization transforms this sequence into a bit string of length at least

$$\sum_{a \in [0..\sigma)} f_a \cdot \log \frac{n}{f_a} + \frac{\sigma - 1}{4} + r - 2,$$

where $f_a$ is the number of occurrences for symbol $a$. Thus, the redundancy is $\frac{\sigma - 1}{4} + r - 2$ bits.

The example that attains the lower bound of Theorem 3 is simple (perhaps, unlike its tedious analysis). Hence, it is reasonable to assume that any adequate shuffling method would have the same redundancy $\Omega(\sigma + r)$ as in Theorem 3 on a similarly constructed sequence $a_1, a_2, \ldots, a_n$. We believe, therefore, that Duda’s conjecture that the redundancy can be $O(\frac{\sigma}{\sigma^r})$ bits per symbol when an appropriate shuffling method is used is disproved. This probably is not the case for algorithms that construct the shuffling tables after scanning the sequence first. However, such methods seem infeasible in practice due to incurring performance losses.
V. RANGE ANS WITH FIXED ACCURACY

In Section III-A, a simple range ANS (rANS) was described, which is just the ANS without shuffling. Now we are to introduce another variant of rANS, called rANS with fixed accuracy to distinguish it from the rANS as defined by Duda [14], [16] (which is also sketched below). Our exposition is less detailed than in the previous sections since we believe that all ideas and intuition necessary for understanding were developed above.

Let $a_1, a_2, \ldots, a_n$ be a sequence of symbols over an alphabet $[0..\sigma)$, where $n = 2^r$ and the frequencies of symbols are denoted by $f_a$ (i.e., the empirical probability for symbol $a$ is $\frac{f_a}{\sigma}$). Fix an integer $k \geq 0$, which will serve as a user-defined accuracy parameter regulating the size of redundancy (typically, $0 \leq k \leq 4$). As in the simple rANS, the encoder starts its work with a number $w_0 = 2^{r+k}$; the value $w_0$ might be arbitrary, the only necessary condition is $w_0 \geq 2^{r+k}$. The encoder consecutively performs operations $w_{i+1} = \text{push}(w_i, a_{i+1})$, for $i = 0, 1, \ldots, n-1$, but the operation $\text{push}(w, a)$ works differently this time: it substitutes either $\lfloor \log f_a \rfloor + k + 1$ or $\lfloor \log f_a \rfloor + k + 2$ highest bits of $w$ that store a number $x$ by $r+k+1$ new bits that store the number $x' = [x/f_a]2^r + c_a + (x \mod f_a)$, where $c_a = \sum_{b\in[0..a]} f_b$; the condition determining the number of bits occupied by $x$ is essentially the same as in the simple rANS from Section III-A.

More formally, the algorithm for $\text{push}(w, a)$ is as follows. Denote by $x_1$ and $x_2$ the values stored in, respectively, the highest $\lfloor \log f_a \rfloor + k + 1$ and $\lfloor \log f_a \rfloor + k + 2$ bits of $w$. We assume that $x = x_1$ if $x_1 \geq f_a2^k$, and $x = x_2$ otherwise. Since $2^{\lfloor \log f_a \rfloor + k + 1} \leq x_1 < f_a2^{k+1}$ and $x_2 \leq 2 \cdot x_1 + 1$, we have $f_a2^k \leq x < f_a2^{k+1}$; Denote $\ell = \lfloor \log w \rfloor - \lfloor \log x \rfloor$. Note that $w = x \cdot 2^\ell + (w \mod 2^\ell)$. The value $w' = \text{push}(w, a)$ is computed as $w' = x' \cdot 2^\ell + (w \mod 2^\ell)$ by replacing the part $x$ of $w$ by $r+k+1$ bits representing a number $x'$ defined as:

$$x' = [x/f_a]2^r + c_a + (x \mod f_a).$$  \hspace{1cm} (10)

Since $f_a2^k \leq x < f_a2^{k+1}$, we have $2^k \leq [x/f_a] < 2^{k+1}$ and, therefore, the number $x'$ indeed fits into $r+k+1$ bits and its highest $(r+k+1)$th bit is 1. The reverse operation $(w, a) = \text{pop}(w')$ producing the old value $w$ and the symbol $a$ from $w'$ is straightforward. Given a number $x'$, which occupies $r+k+1$ highest bits of $w'$ (i.e., $x' = [w'/2^\ell]$), where $\ell = \lfloor \log w' \rfloor - \lfloor \log x \rfloor$, we first determine the symbol $a$ by examining to which range $[c_a .. c_{a+1}]$ the number $x' \mod 2^\ell$ belongs and, then, we compute $x$ as follows:

$$x = [x'/2^\ell]f_a + (x' \mod 2^\ell) - c_a.$$  \hspace{1cm} (11)

Once $x$ is known, we put $w = x \cdot 2^\ell + (w' \mod 2^\ell)$. It remains to observe that $x' > x$ since $x' = [x/f_a]2^r + (x \mod f_a) > [x/f_a]f_a + (x \mod f_a) = x$. Therefore, we have $w_0 < w_1 < \cdots < w_n$.

To estimate the size of the final number $w_n$ in bits, we derive by analogy to (6) and (7) using the condition $x' > x$ the following two equations (here we have $w_i = x \cdot 2^\ell + \Delta$ and $w_{i+1} = x' \cdot 2^\ell + \Delta$, where $\Delta \in (0..2^\ell)$):

$$\log w_n = \log \frac{w_n}{w_{n-1}} + \log \frac{w_{n-1}}{w_{n-2}} + \cdots + \log \frac{w_1}{w_0} + r + k;$$

$$\log w_{i+1} - \log w_i \leq \log x' - \log x.$$  \hspace{1cm} (12)

From (11), we deduce $\log x \geq \log ([x'/2^\ell]f_a) \geq \log f_a + \log(x'/2^\ell - 1) = \log f_a + \log(x'/2^\ell) + \log(1-2^\ell/x') = \log f_a + \log x' + \log(1-2^\ell/x')$. Since $2^{r+k} \leq x'$, we derive further $\log(1-2^\ell/x') \geq \log(1-1/2^k) \geq -\frac{\log x}{2^k-1}$. Thus we obtain

$$\log x' - \log x \leq \log \frac{n}{f_a} + \frac{\log e}{2^k-1}. $$

Summing the values $\log x' - \log x$ over all $n = 2^r$ symbols of the sequence, we obtain the following theorem.

**Theorem 4:** Given a sequence $a_1, a_2, \ldots, a_n$ of symbols from an alphabet $[0..\sigma)$ such that each symbol $a$ occurs in it $f_a$ times and $n = 2^r$ for an integer $r$, the rANS encoder with fixed accuracy $k \geq 1$ transforms this sequence into a bit string of length

$$\sum_{a\in[0..\sigma]} f_a \cdot \log \frac{n}{f_a} + O\left(\frac{n}{2^k} + r\right),$$

where the $O\left(\frac{n}{2^k} + r\right)$ redundancy term can be bounded by $\frac{n \log e}{2^k} + r$.

For completeness, let us briefly describe the standard rANS [14], [16]. The encoder similarly starts with the number $w_0 = 2^r$ and consecutively computes $w_1, w_2, \ldots, w_n$ for the sequence $a_1, a_2, \ldots, a_n$, where $n = 2^r$. The encoder maintains a number $t$ (initially $t = r + 1$) and, receiving a new symbol, it replaces the highest $t$ bits of the current number $w_t$ with a larger value and increases $t$ accordingly. A “renormalization” is sometimes performed by reducing the value $t$ in order to contain numbers within a range fitting into a machine word. More precisely, receiving a symbol $a = a_{i+1}$, the encoder takes the value $x$ stored in the $t$ highest bits of $w_t$ (i.e., $w_t = x \cdot 2^\ell + \Delta$, where $\ell = \lfloor \log w_t \rfloor + 1 - t$ and $\Delta \in (0..2^\ell)$), calculates a number $x'$ by formula (10), and replaces $x$ with $x'$, thus producing $w_{i+1} = x' \cdot 2^\ell + \Delta$. After this, the value $t$ is increased by $\lfloor \log x' \rfloor - \lfloor \log x \rfloor$. Once $t$ is larger than a fixed threshold $T$, it is decreased but the resulting $t$ should be larger than $r$. 
Usually, for performance reasons, encoders decrease $t$ by a multiple of 8 or 16: $t = r + 1 + (t - r - 1 \mod b)$, where $b = 8$ or $b = 16$. Implementations maintain the $t$-bit number $x$, assigning $x = x'$ after processing each symbol, and the decreased number of bytes from $x$ are “dumped” into an external stream. The decoder executes the same operations but in the reverse order computing $x$ from $x'$ as in (11).

The analysis of this rANS variant is not in the scope of the present paper; see [14], [33].

a) Implementation notes: In our experiments the described rANS encoder with fixed accuracy $k = 3$ was two times slower than the tANS encoder implemented by Collet [9] and had approximately the same compression rate. It is unsurprising since the inner encoding loop of TANS computing (2) essentially consists of just a couple of accesses to tables stored in the L1 cache. However, the rANS with fixed accuracy is faster than one could have expected from the standard rANS. The key feature that allows the speed boost is that the operation of division $\lfloor x/f_a \rfloor$ in (10) guarantees that its result is in the range $[2^k..2^{k+1})$. Therefore, the division can be executed by simpler instructions in a branchless code: we used arithmetic and bit operations and the instruction cmov from x86 (it is also possible to use only arithmetic and bit instructions). The code, however, turns out to be quite cumbersome, which noticeably diminishes positive effects of the division-free branchless loop.

Denote $R = r + k$. The main loop of the encoder calls the function encode from Algorithm 2 consecutively for the symbols $a_1, a_2, \ldots, a_n$ in the encoded sequence. The function receives as its parameters an $(R+1)$-bit number $w$ and a symbol $a$. The function stores some lowest bits of $w$ in an external storage and returns an $(R+1)$-bit value $x'$ computed as in (10) (details follow). The parameter $w$ is actually an $(R+1)$-bit number $x'$ produced by the previous call to the function encode in the encoding loop; the first call receives $w = 2^R$.

Algorithm 2 The encoding function of rANS with fixed accuracy $k = 3$.

```plaintext
1: function encode(w, a)
2:   (c_a, f_a, d) = table[a]; ▷ rANS with fixed accuracy $k = 3$
3:   s = (w + d) >> (R + 1);
4:   outBits(w, s);
5:   x = w >> s;
6:   x = x - (f_a << 3);
7:   q = 0;
8:   for $i = 2, 1, 0$ do ▷ the loop must be unrolled
9:     $x_0 = x - (f_a << i)$;
10:    if ($x_0 \geq 0$) $x = x_0$;
11:    $q = q \text{ or } (x_0 \text{ and } (1 << (R + i)))$;
12:   return $((q \text{ xor } (15 << R)) >> k) + c_a + x$;
```

Denote $t = r - \lfloor \log f_a \rfloor$. The number $x$ occupies either $R - t + 1$ or $R - t + 2$ highest bits of $w$. The presented pseudocode uses Collet’s trick [9] to determine $x$ with a branchless code. To this end, the array table stores, for each symbol $a \in [0..\sigma)$, besides the values $c_a$ and $f_a$, the number $d = (t << (R + 1)) - (f_a << (t + k))$. The trick is that, in this case, the number $w + d = (t << (R + 1)) + w - (f_a << (t + k))$ contains in its highest bits $R + 1, R + 2, \ldots$ either the number $t$ or $t - 1$ depending on whether $w \geq (f_a << (t + k))$ or not. Therefore, $x = w >> s$, where $s = (w + d) >> (R + 1)$.

The code in lines [6][11] accumulates the quotient $[x/f_a]$ in the variable $q$ and the remainder $x \mod f_a$ in the variable $x$. It is done by subtracting the numbers $f_a << i$, for $i = 3, 2, 1, 0$, from $x$, thus, reconstructing $q$ bit by bit; note, however, that the bits in $q$ are inverted and, hence, in the end we have to perform xor with 15 (11112 in binary).

The standard rANS has a couple of advantages over tANS in some use cases. It does not require a table of size $2^r$ like tANS and, hence, is more convenient for the (pseudo) adaptive mode when the table of frequencies is sometimes rebuilt during the execution of the encoder; due to this less heavy use of memory, the rANS might be better for interleaving several streams of data and utilizes more efficiently the instruction-level parallelism for this task. For these reasons, the rANS was used in some high performance compressors. The described rANS with fixed accuracy shares the same good features of the standard rANS plus the described above benefits of the controlled division. In addition, we believe that the rANS with fixed accuracy can potentially have more efficient hardware implementations using a parallelization for the computation of the division, which is possible since the resulting quotient is in a small range $[2^k..2^{k+1})$.

VI. Conclusion and Open Problems

Theorems 1 and 3 describe the tight asymptotic behaviour of the redundancy for the tANS. We believe that Theorem 4, albeit not complemented with a lower bound, is asymptotically tight too. However, it is open to provided a series of examples supporting this claim. A number of other problems listed below still remain open too.

---

2In [19] Giesen showed that the so-called interleaving streams can significantly speed up the rANS. We did not use this trick, which may speed up our implementation.
(i) The main remaining open problem concerning ANS encoders, as we see it, is to construct a genuine FIFO encoder with the same performance characteristics as tANS or rANS. The known ANS variants work as stacks (LIFO) while it is more natural and, in some scenarios, preferable to have an encoder that acts as a queue, like the arithmetic coding that fulfills this requirement but is noticeably worse than ANS in performance terms. For the same reason, the current ANS variants are not suitable enough for the adaptive encoding when frequencies of symbols change as the algorithm reads the sequence from left to right.

We note that the rANS can be applied for the adaptive mode as follows (for instance, see [2]): it first performs one left-to-right pass on the sequence to collect individual statistics per symbol adaptively and, then, encodes the sequence from right to left using these statistics; the decoder decodes the sequence from left to right adaptively computing the statistics by the same algorithm as was used by the encoder in the first pass. At first glance, this scheme is not much different from what the tANS or non-adaptive rANS do (note that they also perform a separate pass collecting statistics). Unfortunately, the adaptive rANS cannot utilize many optimizations that make a modern rANS comparable to tANS in terms of speed (in particular, the decoder cannot use SIMD and interleaving streams [19] as freely and cannot speed up divisions with special precomputed values). One of the goals for a FIFO encoder is to fix this issue.

(ii) Duda’s precise initialization and its simplified variant from Algorithm 1 are not particularly suitable for practice due to the overhead incurred by operations on the priority queue and by floating point operations. Therefore, usually they are replaced with heuristics. An implementation of a fast and simple initialization method with good guarantees sufficient for Theorem 1 is an open problem.

(iii) The constants in our lower and upper bounds in Theorems 1 and 3 do not coincide and it remains open to find a tight constant for the tANS redundancy term. We believe also that the r-bit redundancy produced by the initial state $w_0$ of the encoder can be somehow reduced too by slightly modifying the scheme. Clearly, one can omit the trailing zeros in the bit representation of the resulting number $w_n$ but it does not suffice to get rid of the r-bit redundancy entirely.

(iv) The encoding of the table of frequencies is rarely in the focus of theoretical research. However, it is very important in practice, mainly due to the following observation: a typical sequence fed to the entropy encoder is not homogeneous but consists of “chunks” drawn from different distributions; accordingly, an optimized compressor normally either uses an adaptive encoder (which seems best for this case but such encoders currently have issues described above) or splits the sequence into blocks and encodes each block separately, each with its own frequency table (e.g., see [3]). On relatively small chunks of data such as those produced in the latter case, the size of the frequency table is not negligible compared to the entropy of the data. For this reason, many practitioners resort to simpler Huffman encoders, which require much less space to encode their frequency tables. Surprisingly, this advantage in practice often suffices to compensate for the larger redundancy of Huffman compared to arithmetic or ANS encoders. Note that it is not an exceptional case: for instance, the Huffman encoder is consistently superior to arithmetic or ANS when encoding literals in LZ77 compressors (some LZ77 compressors often cautiously utilize the Huffman encoders in other parts as well, like to encode LZ77 phrase offsets and phrase lengths, without losses in the compression ratio; however, in most cases these parts still are processed by ANS or arithmetic encoders). Often the authors of compressors resort to the Huffman encoders after extensive experiments with ANS and arithmetic encoders, so the reason is not in simplicity.

Theoretical tight bounds are known for the size of encodings of the frequency tables: see [30], [27], [32], and references therein. However, works that investigate practical encoders for these tables are scarce. Ideally, such an encoder should take into account the entropy of the data to be able to relax some frequencies in order to make the total encoding size smaller, if necessary, thus never being worse than the Huffman encoders (see [15] and references in [28]). It seems that further investigations in this direction are needed.

ACKNOWLEDGMENTS

Supported by the Russian Science Foundation (RSF), project 24-71-00062.
[11] T. M. Cover, J. A. Thomas, Information theory and statistics, Elements of Information Theory 1 (1) (1991) 279–335.
[12] D. Dubé, H. Yokoo, Fast construction of almost optimal symbol distributions for asymmetric numeral systems, in: 2019 IEEE International Symposium on Information Theory (ISIT), IEEE, 2019.
[13] J. Duda, Asymmetric numeral systems, arXiv preprint arXiv:0902.0271 (2009) 1–47.
[14] J. Duda, Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding, arXiv preprint arXiv:1311.2540 (2013) 1–24.
[15] J. Duda, Encoding of probability distributions for asymmetric numeral systems, arXiv preprint arXiv:2106.06438 (2021) 1–5.
[16] J. Duda, K. Tahboub, N. J. Gadgil, E. J. Delp, The use of asymmetric numeral systems as an accurate replacement for Huffman coding, in: 2015 Picture Coding Symposium (PCS 2015), IEEE, 2015.
[17] T. Gagie, Large alphabets and incompressibility, Information Processing Letters 99 (6) (2006) 246–251.
[18] F. Giesen, The ryg blog, https://fgiesen.wordpress.com, accessed: 07.01.2022.
[19] F. Giesen, Interleaved entropy coders, arXiv preprint arXiv:1402.3392 (2014) 1–16.
[20] D. A. Huffman, A method for the construction of minimum-redundancy codes, Proc. Institute of Radio Engineers (IRE) 40 (9) (1952) 1098–1101.
[21] S. R. Kosaraju, G. Manzini, Compression of low entropy strings with Lempel–Ziv algorithms, SIAM Journal on Computing 29 (3) (1999) 893–911.
[22] G. Manzini, An analysis of the Burrows–Wheeler transform, Journal of the ACM 48 (3) (2001) 407–430.
[23] G. N. N. Martin, Range encoding: an algorithm for removing redundancy from a digitised message, in: Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording, 1979.
[24] B. McMillan, Two inequalities implied by unique decipherability, IRE Transactions on Information Theory 2 (4) (1956) 115–116.
[25] A. Moffat, R. M. Neal, I. H. Witten, Arithmetic coding revisited, ACM Transactions on Information Systems 16 (3) (1998) 256–294.
[26] A. Moffat, M. Petri, Large-alphabet semi-static entropy coding via asymmetric numeral systems, ACM Transactions on Information Systems 38 (4) (2020) 1–33.
[27] A. Orlitsky, N. P. Santhanam, Speaking of infinity [iid strings], IEEE Transactions on Information Theory 50 (10) (2004) 2215–2230.
[28] A. Painsky, S. Rosset, M. Feder, Large alphabet source coding using independent component analysis, IEEE Transactions on Information Theory 63 (10) (2017) 6514–6529.
[29] J. J. Rissanen, Generalized Kraft inequality and arithmetic coding, IBM Journal of research and development 20 (3) (1976) 198–203.
[30] Y. M. Shtar'kov, Universal sequential coding of single messages, Problems of Information Transmission 23 (3) (1987) 3–17.
[31] T. Strutz, Rescaling of symbol counts for adaptive rANS coding, in: 2023 31st European Signal Processing Conference (EUSIPCO), IEEE, 2023.
[32] W. Szpankowski, M. J. Weinberger, Minimax pointwise redundancy for memoryless models over large alphabets, IEEE Transactions on Information Theory 58 (7) (2012) 4094–4104.
[33] J. Townsend, A tutorial on the range variant of asymmetric numeral systems, arXiv preprint arXiv:2001.09186 (2020) 1–11.
[34] N. Wang, C. Wang, S.-J. Lin, A simplified variant of tabled asymmetric numeral systems with a smaller look-up table, Distributed and Parallel Databases 39 (3) (2021) 711–732.
[35] H. Yamamoto, K. Iwata, Encoding and decoding algorithms of ANS variants and evaluation of their average code lengths, arXiv preprint arXiv:2408.07322.
[36] H. Yokoo, On the stationary distribution of asymmetric binary systems, in: 2016 IEEE International Symposium on Information Theory (ISIT), IEEE, 2016.
[37] H. Yokoo, D. Dubé, Asymptotic optimality of asymmetric numeral systems, in: Proc. 42nd International Symposium on Information Theory and Its Applications (ISITA 2019), 2019.
[38] H. Yokoo, T. Shimizu, Probability approximation in asymmetric numeral systems, in: Proc. 2018 International Symposium on Information Theory and Its Applications (ISITA 2018), IEEE, 2018.