Fast Construction of Almost Optimal Symbol Distributions for Asymmetric Numeral Systems

Danny Dubé
Department of Computer Science and Software Engineering
Université Laval, Canada
Email: Danny.Dube@ift.ulaval.ca

Hidetoshi Yokoo
Division of Electronics and Informatics
Gunma University, Japan
Email: Yokoo@gunma-u.ac.jp

***Abstract*—A crucial task in the design of an efficient ANS encoder consists in choosing a favourable symbol distribution. This task seems to be hard, due to its combinatorial nature, in particular for the tANS variant of ANS, which is the focus of this work. We present a fast technique that builds almost optimal symbol distributions for the stream variant of tANS.**

I. Introduction

Asymmetric numeral systems (ANS) have been proposed by Duda [2], [3] and they start to attract more and more attention. ANS is an entropy-coding technique that combines the speed of Huffman coding with the performance of arithmetic coding, in terms of redundancy. ANS has been used in various applications [4], [5], [8]–[11]. It has also been theoretically studied under different angles: its redundancy [12]–[14], the characteristics of its uniform variant [6], [7], and the construction of the symbol distributions it is based on [1]. In this paper, we propose a fast technique that allows one to build very good, yet not always optimal, symbol distributions for the stream variant of ANS.

II. Background on ANS

Here, we present a brief introduction to ANS. First, we present ANS with unbounded states. Second, we present the variant that manipulates a finite number of states, which is called *stream ANS*. Third, we describe how to determine the average codeword length (ACL) of a stream ANS encoder. Finally, we mention various existing techniques to construct the so-called symbol distributions.

A. Basics of ANS

In essence, ANS is an entropy coding technique designed for a stationary and memoryless random source over a finite alphabet. Let $\mathcal{A}$ be the source alphabet. The symbol probabilities are assumed to be known. Let $p(s)$ denote the probability of $s \in \mathcal{A}$. Duda calls the technique Asymmetric Binary Systems (ABS) when $\mathcal{A}$ is binary and ANS, otherwise. In this paper, without loss of generality, we choose to systematically call the technique ANS.

An ANS encoder processes a sequence of symbols from $\mathcal{A}$ by updating a state, which is simply a natural number. The more symbols get encoded, the larger the state becomes. The encoding of a symbol is made injectively, in the following sense. Let the current state be $x \in \mathbb{N}$ and the symbol that gets encoded be $s \in \mathcal{A}$, resulting in the new state $x' \in \mathbb{N}$, then x' uniquely determines x and s. When the ANS encoder is used to encode a complete string $w \in \mathcal{A}^*$, then the encoder starts with the initial state x_I, encodes each symbol of w successively, and finally transmits the state x_F that has ultimately been reached. The initial state x_I is a predetermined state in the communication protocol between the ANS encoder and the corresponding ANS decoder. Thanks to the injective mapping, the ANS decoder is able to recover each symbol of w successively, *from the last to the first*, starting from x_F and decoding symbols until it reaches x_I.

The most crucial element in the design of an ANS encoder-decoder pair is the *symbol distribution* $\overline{s}$. In this paper, we rather refer to $\overline{s}$ as the *key* of the encoder-decoder pair. Key $\overline{s}$ is an infinite string on $\mathcal{A}$. Minimally, $\overline{s}$ has to obey the property that it contains an infinite number of each symbol of $\mathcal{A}$. Figure 1 shows an example of a key for the alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. When the encoder is in state x, encodes symbol s, and makes a transition to state x', then x' is the position of the $(x+1)$st occurrence of s in $\overline{s}$. Note that we count positions from zero. If we come back to the example in Figure 1 and assume that symbol 'b' gets encoded, then we have that: if x is 0, then x' is 1; if x is 1, then x' is 5; if x is 2, then x' is 8; and so on. This way of encoding symbols is injective because of the following two reasons: first, s can be recovered from x' since we have $\overline{s}(x') = s$; second, x can be recovered since $\overline{s}(x')$ is the $(x+1)$st occurrence of s in $\overline{s}$.

The encoding of a symbol by the ANS encoder and the inverse operation of decoding a symbol can be easily expressed using the 'rank' and 'select' functions. These functions are well known in computer science, especially in string-manipulation algorithms and in compressed data structures.

$$\begin{aligned} C &: \mathcal{A} \times \mathbb{N} \to \mathbb{N} \\ C(s, x) &= \mathrm{select}_s(x, \overline{s}) \qquad \text{// counting from 0} \\ D &: \mathbb{N} \to \mathcal{A} \times \mathbb{N} \\ D(x) &= (s, \mathrm{rank}_s(x, \overline{s})), \qquad \text{// counting from 0} \\ &\quad \text{where } s = \overline{s}(x) \end{aligned}$$

B. Stream Variant of ANS

The main problem with the basic ANS technique is the need to manipulate large numbers. In order to solve this problem,

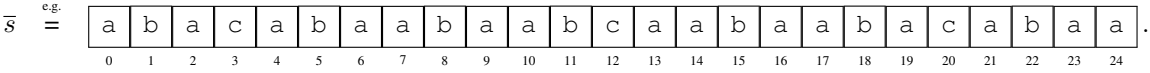

Fig. 1. Example key from Duda'14 [3].

Duda proposed a variant called *stream ANS* that constrains the states to lie in a chosen interval I of natural numbers.

Since the encoding of symbols causes the current state to grow, stream ANS includes an additional mechanism that flushes bits out of the current state in order to reduce its magnitude. Duda presented a general setting in which the output alphabet has arbitrary size b and the reduction of the magnitude of the current state is performed by emitting base b numbers. In this paper, we consider only the binary output alphabet; i.e. bits get extracted out of the current state and their emission forms the compressed data.

Choosing I reduces to selecting its smallest element l:

$$I \;=\; \{l, l+1, \ldots, 2l-1\}\ .$$

Since encoding a symbol s when the encoder is in state $x \in I$ might cause the updated state $x' = C(s, x)$ to lie beyond I. In order to prevent such an overflow, safe *pre-images* are determined for each symbol $s \in \mathcal{A}$:

$$I_s \;=\; \{x \in \mathbb{N} \mid C(s, x) \in I\}\ .$$

These pre-images are intervals, like I. They are not contained in I but they may have a non-empty intersection with I. Let l_s be the smallest element in I_s, for each $s \in \mathcal{A}$. Each pre-image I_s must have a specific length:

$$I_s \;=\; \{l_s, l_s+1, \ldots, 2l_s-1\}\ .$$

The constraints on the length of I and every I_s make all of these 2-*absorbing*.[1] The fact that I is 2-absorbing means that, for any $n \in \mathbb{N} - \{0\}$, there exists a unique $k \in \mathbb{Z}$ such that $\lfloor 2^k n \rfloor \in I$; similarly for every I_s.

The encoding function for the stream variant makes sure to flush bits out of the current state prior to really encoding the prescribed symbol. It returns both the bits to emit and the updated state, which is guaranteed to lie in I.

$$\begin{array}{rcl} \vec{C} & : & \mathcal{A} \times \mathbb{N} \to \{0,1\}^* \times I \\ \vec{C}(s, x) & = & \begin{cases} (\epsilon, C(s, x)), & \text{if } x \in I_s \\ ((x \bmod 2) \cdot v, x'), & \text{otherwise} \\ \qquad \text{where } (v, x') = \vec{C}\left(s, \lfloor \frac{x}{2} \rfloor\right) \end{cases} \end{array}$$

Note that, in the type of $\vec{C}$, we write $\mathbb{N}$ instead of I as the type of the second argument. This is because $\vec{C}$ may call itself recursively with state numbers that get below I.[2]

The decoding function $\vec{D}$ for the stream variant basically acts like D, except that, after decoding the symbol s from the current state $x \in I$ and recovering the smaller pre-state $x' \in I_s$, it takes care of enlarging x' with compressed bits in order to inflate it up to $x'' \in I$.

$$\begin{array}{rcl} \vec{D} & : & I \to \mathcal{A} \times I \\ \vec{D}(x) & = & \left(s, \vec{D}_{\text{aux}}(x')\right), \text{ where } (s, x') = D(x) \\ \vec{D}_{\text{aux}} & : & \mathbb{N} \to I \\ \vec{D}_{\text{aux}}(x) & = & \begin{cases} x, & \text{if } x \in I \\ \vec{D}_{\text{aux}}(2x + \textbf{receive\_bit}()), & \text{otherwise} \end{cases} \end{array}$$

Note that this definition of $\vec{D}$ is not as clean as its inverse $\vec{C}$. This is because $\vec{D}$ uses the function '**receive_bit**', which causes a side effect by reading a bit from the compressed data. Providing a definition of $\vec{D}$ that is free of side effects would be messy. It would have to receive the whole sequence of compressed bits, extract some prefix, depending on its needs, and return the rest of the bit sequence. We believe that it is slightly more elegant to use a function with side effects.

Example. Let us present the example of a stream ANS encoder. The encoder is designed for $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and key $\overline{s}$ as shown in Figure 1. Let us suppose that we choose I to be $\{7, 8, \ldots, 13\}$. Note that I is 2-absorbing. We must determine $I_{\mathtt{a}}$, $I_{\mathtt{b}}$, and $I_{\mathtt{c}}$. The occurrences of '`a`' that I contains are the 5th to the 8th. So, $I_{\mathtt{a}}$ is $\{4, 5, 6, 7\}$. In a similar fashion, we can determine that $I_{\mathtt{b}}$ is $\{2, 3\}$ and $I_{\mathtt{c}}$ is $\{1\}$. Note that all of $I_{\mathtt{a}}$, $I_{\mathtt{b}}$, and $I_{\mathtt{c}}$ are 2-absorbing.[3] Finally, we determine how the encoder proceeds when it is in state x and it encodes s, for all $s \in \mathcal{A}$ and $x \in I$. Figure 2 shows all the possible cases. Each cell of the table indicates, in this order: the bits that get flushed out by reducing x down into I_s, the reduced state thus obtained, and $\vec{C}(s, x)$.

In order to provide a more detailed explanation about this example, we describe what happens when the encoder is in state $x = 13$ and symbol $s = \mathtt{b}$ gets encoded. First, we note that x is too high to be in $I_{\mathtt{b}}$. So a bit must get flushed out of x. This bit is the parity bit of x, which is `1`. The reduced state is $\lfloor \frac{13}{2} \rfloor = 6$. Still, 6 is too high to be in $I_{\mathtt{b}}$. So another bit must get flushed out of 6. This second bit is the parity bit of 6, which is `0`. The once again reduced state is 3, which lies in $I_{\mathtt{b}}$. Now, the symbol can effectively be encoded into $C(\mathtt{b}, 3) = 11$. To summarize the contents of the cell in the table, the current state x was too high to lie in $I_{\mathtt{b}}$, so by flushing the two bits `10` (in that order), we obtain the reduced state $3 \in I_{\mathtt{b}}$, which gets mapped to 11 by encoding '`b`'.

[1] When a more general output alphabet of size b is considered, then all these intervals have to be b-absorbing; i.e. $I = \{l, l+1, \ldots, b \cdot l - 1\}$ and $I_s = \{l_s, l_s+1, \ldots, b \cdot l_s - 1\}$, for every $s \in \mathcal{A}$.

[2] Note that these smaller state numbers cannot get below l_s, when $\vec{C}$ encodes symbol s, because I_s is 2-absorbing. More specifically, in calls like $\vec{C}(s, x)$, x is guaranteed to lie in $\{l_s, l_s+1, \ldots, 2l-1\}$.

[3] As a technical note, we point out that the choice of I cannot be made arbitrarily, since some choices would induce pre-images that are not 2-absorbing. The particular I used in this example has been chosen appropriately.

$s\backslash x$	7	8	9	10	11	12	13
a	$\frac{\epsilon}{7}{13}$	$\frac{0}{4}{7}$	$\frac{1}{4}{7}$	$\frac{0}{5}{9}$	$\frac{1}{5}{9}$	$\frac{0}{6}{10}$	$\frac{1}{6}{10}$
b	$\frac{1}{3}{11}$	$\frac{00}{2}{8}$	$\frac{10}{2}{8}$	$\frac{01}{2}{8}$	$\frac{11}{2}{8}$	$\frac{00}{3}{11}$	$\frac{10}{3}{11}$
c	$\frac{11}{1}{12}$	$\frac{000}{1}{12}$	$\frac{100}{1}{12}$	$\frac{010}{1}{12}$	$\frac{110}{1}{12}$	$\frac{001}{1}{12}$	$\frac{101}{1}{12}$

Fig. 2. Transition table of an example stream ANS encoder.

C. Measuring the Efficiency of Stream ANS

The efficiency of an ANS encoder is measured in terms of its ACL. In order to compute the ACL of a stream encoder, one first has to compute the stationary probability distribution $\mathcal{P}$ on the states of I, where $\mathcal{P} : I \to (0, 1)$. Then one computes the expected length $L(\mathcal{P})$ of the bit sequence that gets output when encoding a randomly chosen source symbol.

As shown in Figure 2, the length of the bit sequence that gets output varies depending on the symbol and the state, but it does so by at most one bit, for any given symbol. In fact, if we let λ_s be $\lceil -\log_2(l_s/l) \rceil$ and ξ_s be $l_s 2^{\lambda_s}$, then we can show that, when $l \leq x < \xi_s$, the length of the codeword for symbol s is $\lambda_s - 1$ bits and, when $\xi_s \leq x < 2l$, it is λ_s bits [1], [3], [4], [7]. Then, the ACL is given by:

$$\begin{aligned} L(\mathcal{P}) &= \sum_{s\in\mathcal{A}} p(s) \left((\lambda_s - 1) \sum_{x=l}^{\xi_s - 1} \mathcal{P}(x) + \lambda_s \sum_{x=\xi_s}^{2l-1} \mathcal{P}(x) \right) \\ &= \sum_{s\in\mathcal{A}} p(s)\lambda_s - \sum_{s\in\mathcal{A}} p(s) \sum_{x=l}^{\xi_s - 1} \mathcal{P}(x) \end{aligned} \quad (1)$$

Equation (1) suggests that it is beneficial to have more probability mass on the left side in $\mathcal{P}$. The technique that we propose in Section III is based on these observations.

D. Construction of Keys

Duda proposed three techniques for constructing keys. We point out that, in each technique, the number of states that are granted to each symbol of $\mathcal{A}$ tends to be proportional to the symbol's probability. The difference comes from the way specific states get associated to symbols.

1) Uniform ANS: Uniform ANS is a key-construction technique that tries to spread occurrences of the symbols in the key as evenly as possible, according to the probabilities. For example, a symbol with probability $1/3$ would be placed roughly once every three positions in the key. We write "roughly" because the needs of the other symbols to be spread as evenly as possible conflict with this symbol's needs. Figure 1 shows a key that could have been built using uANS.

In the case of a binary alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$, Duda could give a direct formula that determines $\overline{s}(x)$, for all $x \in \mathbb{N}$. According to the formula, x is associated to '$\mathtt{b}$' if and only if $\lceil (x+1) \cdot p(\mathtt{b}) \rceil - \lceil x \cdot p(\mathtt{b}) \rceil = 1$. Duda could also derive direct formulas for C and D. These allow one to avoid the explicit construction of the key and the use of 'rank' and 'select'. Such formulas have been devised only for the case $|\mathcal{A}| = 2$, which justifies the use of a specific name for the ANS technique in this case: asymmetric *binary* systems (ABS). The construction in a uniform way is referred to as uABS. The existence of the direct formulas makes uABS adequate for use in both stream and non-stream variants of ANS.

On the other hand, when $|\mathcal{A}| > 2$, there are no direct formulas that have been derived for an analogous uANS technique. So, $\overline{s}$ cannot be represented implicitly using a direct formula. The key would have to be built explicitly using an algorithm that would allocate states to symbols, while attempting to spread them evenly. Also, the use of 'rank' and 'select' would be mandatory. In that respect, uANS would not be adequate for the non-stream variant of ANS.

2) Tabled ANS: Tabled ANS (tANS) is a construction technique that consists in constructing a *segment* of the key and defining the key as the concatenation of infinitely many copies of that segment. Let k be the length of the segment. The contents of the segment must be selected explicitly and stored. Direct formulas can be obtained which, given a state x, manipulate the number of the segment copy that contains x, $\lfloor \frac{x}{k} \rfloor$, and the position of x inside the segment, $x \bmod k$. The details about the contents of the segment, like the symbols that are associated to the positions in the segment, are kept in tables, hence the name of the tANS technique. It is adequate to build the key using tANS in both stream and non-stream variants of ANS. The memory footprint of tANS's tables is modest. By carefully constructing the segment, tANS can lead to efficient encoders. The key shown in Figure 1 could also have been built using tANS, with a segment of size 17.

The main difficulty in constructing the segment is that it is a combinatorial task. The segment may be any string in $\mathcal{A}^k$ such that every symbol of $\mathcal{A}$ appears at least once. In principle, we can decompose the construction of a segment into two operations: choosing a size-k bag of symbols from $\mathcal{A}$; selecting an order for these symbols. The bag specifies how many times s appears in the segment, for each $s \in \mathcal{A}$. From now on, the numbers of occurrences of the symbols in the bag are collectively called the *type* of the segment. If $\mathcal{A}$ is $\{s_1, \ldots, s_{|\mathcal{A}|}\}$, then a type is a tuple $(k_1, \ldots, k_{|\mathcal{A}|})$ of strictly positive integers such that $\sum_{i=1}^{|\mathcal{A}|} k_i = k$. So, using standard combinatorics, we can determine that there are $\binom{k-1}{|\mathcal{A}|-1}$ ways to choose the type and, given a type, there are $\binom{k}{k_1,\ldots,k_{|\mathcal{A}|}}$ ways to order the symbols in the segment. It is especially the selection of an order given a type, rather than the choice of a type, that leads to a large search space. In the state of the art, there is no evidence yet that building an optimal key takes polynomial time, in the worst case.

3) Range ANS: The construction technique called *range ANS* (rANS) shares with tANS the fact that the key is made of an infinite concatenation of a segment. The difference comes from the arrangement of the symbols in the segment: these

appear in lexicographic order. As a consequence, the type directly determines the segment: $s_1^{k_1} s_2^{k_2} \dots s_{|\mathcal{A}|}^{k_{|\mathcal{A}|}}$. The segment is made of $|\mathcal{A}|$ runs (or ranges), hence the name of rANS. An advantage of rANS is that the encoder only has to keep a table about the segment's type, which has size $\theta(|\mathcal{A}|)$, rather than a table about the segment's actual contents, which has size $\theta(k)$. A disadvantage of rANS comes from the incapacity to perform any tuning on the order of the symbols in the segment. This usually leads to a lower efficiency than tANS, for the same segment size. The first segment that is shown in Figure 3 is one that would typically be built using rANS.

III. Sort-Based Construction

The main contribution of this paper consists in proposing a technique that builds very good keys with little computations. Our technique can be viewed as an instance of the tANS key-construction technique. It defines the segment to have length $k = l = |I|$. Specifically, it offers a way to quickly select an order for the symbols in the segment, given the preliminary choice of l and the segment type.

We call our technique a *sort-based* construction because it proceeds by sorting the symbols occurrences in the segment according to the stationary probabilities of the corresponding states. Given a candidate segment, it computes the stationary probability distribution $\mathcal{P}$ on the states that is induced by the segment. Then, if $\mathcal{P}$ is not such that $\mathcal{P}(l) \geq \mathcal{P}(l+1) \geq \dots \geq \mathcal{P}(2l-1)$, then a new candidate segment is obtained by sorting the symbols according to the stationary probabilities of the corresponding states. By repeating this process, the candidates quickly get closer to having $\mathcal{P}$ sorted. The ACL tends to improve during the process. After a few iterations, some obtained candidate happens to be identical to a previously seen candidate. Then the technique stops and the best of the considered candidates is retained.

Let us illustrate the operations performed by the sort-based construction using an example; see Figure 3. We consider a random source with $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, with $p(\mathtt{a}) = {}^{10}/_{17}$, $p(\mathtt{b}) = {}^{5}/_{17}$, and $p(\mathtt{c}) = {}^{2}/_{17}$, and where I is $\{17, 18, \dots, 33\}$. The initial segment is one that would be selected by the rANS technique. This segment induces $\mathcal{P}$. Interestingly, we can observe that the initial candidate, which has a very regular order of the symbols, induces a $\mathcal{P}$ function that is very irregular. Next, the second candidate is obtained by assigning symbols to the states of I by following the order set by $\mathcal{P}$ in the first segment. In details, the maximal value $\mathcal{P}(x)$ is obtained with $x = 17$ and 17 is associated to 'a' in the initial candidate, so the second candidate starts with 'a'. The state x' with the second maximal value $\mathcal{P}(x')$ is 27 and 27 is associated to 'b' in the initial candidate, so the second candidate continues with 'b'. The sorting continues up to the full construction of the second candidate. Now, we can observe that $\mathcal{P}$ for the second candidate is much more regular, but not completely sorted. So the technique continues with the construction of another candidate by sorting the symbols again. The third candidate induces a $\mathcal{P}$ that is almost sorted: the only inversion comes from $\mathcal{P}(31) < \mathcal{P}(32)$. Sorting the symbols again brings the fourth candidate. This candidate happens to be the final one. Indeed, $\mathcal{P}$ is sorted, so sorting the segment again would not change it. The ACL is indicated for each candidate. We can observe that the performance improves from one candidate to the next. We point out that we chose $\mathcal{A}$ and $p(\cdot)$ after an example by Duda [3] and the final candidate that we obtain is one of the 32 optimal segments that exist for that random source.

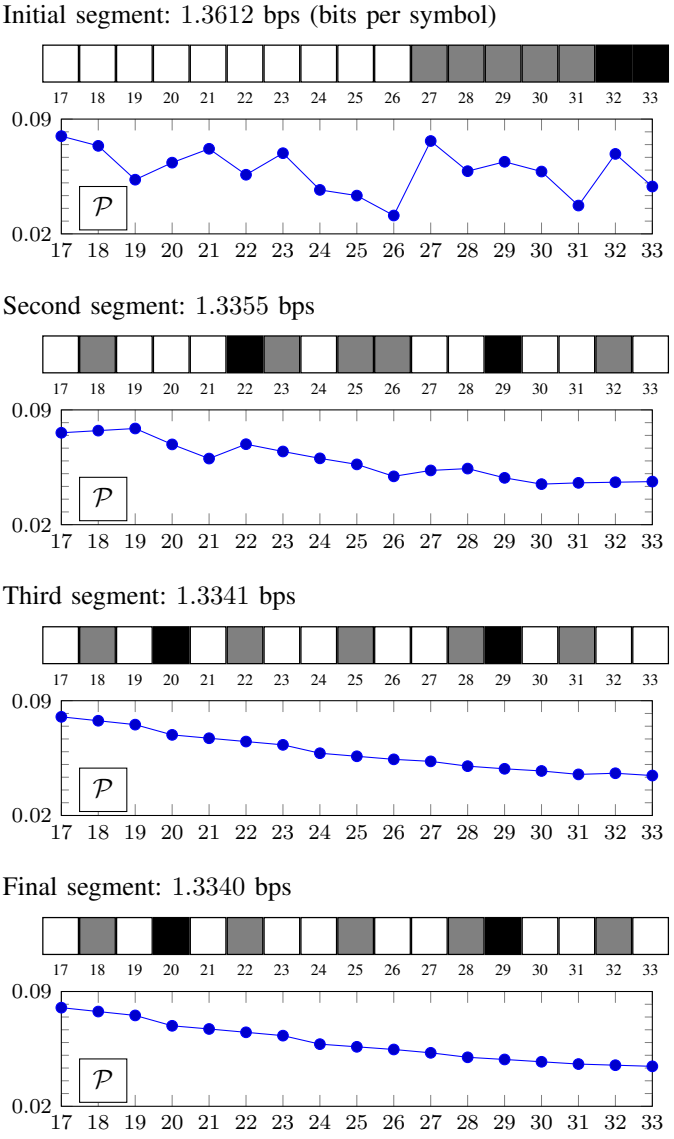

Fig. 3. Sort-based construction of a segment.

IV. Characteristics of the Construction

Here we highlight characteristics of our proposed technique. The existence of nice properties might help to address the hardness of the problem caused by its combinatorial nature. Unfortunately, the problem of constructing optimal keys seems to be inherently hard.

A. Convergence to the Global Minimum

One may wonder whether, given the choice of a segment type and I, the sort-based technique guarantees to converge to the optimal order. We observed that it does not. For example, if we consider the random source of Section III and $I = \{17, 18, \ldots, 33\}$ again and choose a type where there are 13 'a's, 1 'b', and 3 'c's, we observe that the ACL starts at 1.7932 bps, drops to 1.6549 bps, then to 1.6545 bps, and finally goes up to 1.6548 bps.

B. Symbol Frequencies versus Symbol Probabilities

Possibly, one may be satisfied with the pragmatic approach of using the sort-based technique to select an order of the symbols, given a particular type. However, our technique does not provide a solution to the (less hard) problem of choosing a type for the segment. One may wonder if, in the problem of choosing a type, it is correct to assume that a more probable symbol should be assigned more states. Apparently, it seems not to be the case.[4] When we considered the random source such that $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{f}\}$, $p(\mathtt{a}) = {}^{10}\!/_{37}$, $p(\mathtt{b}) = {}^{8}\!/_{37}$, $p(\mathtt{c}) = {}^{7}\!/_{37}$, $p(\mathtt{d}) = {}^{5}\!/_{37}$, $p(\mathtt{e}) = {}^{4}\!/_{37}$, $p(\mathtt{f}) = {}^{3}\!/_{37}$, and $I = \{11, 12, \ldots, 21\}$, the type with 2 'a's and 3 'b's brought the best ACL (along with 2 'c's, 2 'd's, 1 'e', and 1 'f').

C. Interval Size versus Average Codeword Length

One may also wonder if a larger I necessarily improves the ACL. Apparently, it is not the case either. For example, for the random source where $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$, $p(\mathtt{a}) = {}^{7}\!/_{10}$, and $p(\mathtt{b}) = {}^{3}\!/_{10}$, the ACL deteriorates from .88652 bps to .88654 bps, when l goes from 7 to 8.

D. Proximity to the Optimal Average Codeword Length

In order to get an idea of how close to optimality our sort-based technique is, we have run experiments on a family of ternary random sources. The ith source of the family, for $1 \leq i \leq 18$, defines p_1 as $\frac{2i-1}{80}$ and fixes p_2 and p_3, with $p_2 < p_3$, so that the entropy of the source is 1. Interval I has size 11 in all these experiments.

Figure 4 presents the ACLs achieved for the various random sources using different key-construction techniques. For most random sources, our proposed technique builds a key that is virtually optimal, if not optimal. The keys built using uANS are generally less efficient, except in two cases. The keys built using rANS are generally not competitive.

V. Conclusion and Future Work

We propose a technique for building keys (i.e. symbol distributions) for tANS quickly that achieve very good average codeword lengths. The technique uses sorting, based on the stationary probabilities of the states in the stream variant of ANS. Choosing a key can be decomposed into two sub-tasks: choosing a type for a segment of the key and then selecting an order, given that type. In future work, better ways to choose the segment type ought to be devised.

[4]We write that it is "apparently" not the case because we merely use our sort-based technique to select a good order, given the type. It is not like identifying the optimal order, given the type.

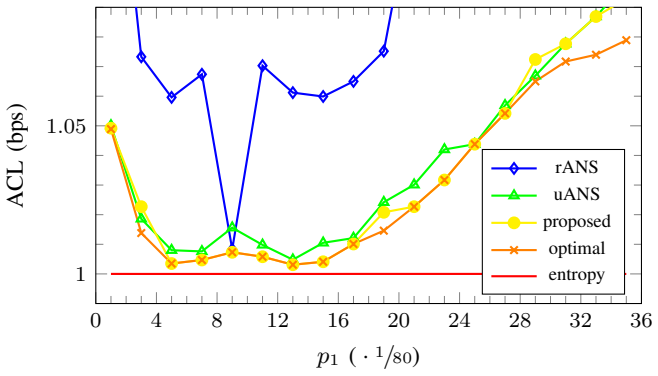

Fig. 4. ACLs obtained on a family of ternary random sources with keys built using different techniques.

Acknowledgment

The first author wishes to thank Jarek Duda for the numerous and extended discussions on ANS. The research of the second author is supported by the JSPS Kakenhi grant number JP17K00004. We also wish to thank the anonymous reviewers of this paper.

References

[1] D. Dubé and H. Yokoo, "Empirical evaluation of the effect of the symbol distribution on the performance of ANS," poster presented at the SITA Symposium, December 2018.

[2] J. Duda, "Asymmetric numeral systems," 2009, arXiv:0902.0271.

[3] ——, "Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding," 2014, arXiv:1311.2540v2.

[4] 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 *Proceedings of the Picture Coding Symposium*, Cairns, Australia, May 2015, pp. 65–69.

[5] J. Duda and M. Niemiec, "Lightweight compression with encryption based on asymmetric numeral systems," 2016, arXiv:1612.04662.

[6] H. Fujisaki, "Invariant measures for the subshifts associated with the asymmetric binary systems," in *Proceedings of the International Symposium on Information Theory and its Applications*, Singapore, October 2018, pp. 675–679.

[7] ——, "On irreducibility of the stream version of the asymmetric binary systems," in *Proceedings of the SITA Symposium*, Iwaki, Fukushima, Japan, December 2018, pp. 218–222.

[8] F. Giesen, "Interleaved entropy coders," February 2014, arXiv:1402.3392v1.

[9] A. Moffat and M. Petri, "ANS-based index compression," in *Proceedings of the International Conference on Information and Knowledge Management*, Singapore, November 2017, pp. 677–686.

[10] ——, "Index compression using byte-aligned ANS coding and two-dimensional contexts," in *Proceedings of the International Conference on Web Search and Data Mining*, Marina del Rey, California, USA, February 2018, pp. 405–413.

[11] 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*, November 2018.

[12] H. Yokoo, "On the stationary distribution of asymmetric binary systems," in *Proceedings of the International Symposium on Information Theory*, Barcelona, Spain, July 2016, pp. 11–15.

[13] ——, "On the stationary distribution of asymmetric numeral systems," in *Proceedings of the International Symposium on Information Theory and its Applications*, Monterey, California, USA, October 2016, pp. 631–635.

[14] H. Yokoo and T. Shimizu, "Probability approximation in asymmetric numeral systems," in *Proceedings of the International Symposium on Information Theory and its Applications*, Singapore, October 2018, pp. 670–674.