Asymmetric numeral systems as close to capacity low state entropy coders

Jarek Duda

Center for Science of Information, Purdue University, W. Lafayette, IN 47907, U.S.A.
email: dudaj@purdue.edu

Abstract

Imagine a basic situation: we have a source of symbols of known probability distribution and we would like to design an entropy coder transforming it into a bit sequence, which would be simple and very close to the capacity (Shannon entropy). Prefix codes are the basic method, defining "symbol→bit sequence" set of rules, usually found using Huffman algorithm. They theoretically allow to reduce the distance from the capacity (ΔH) down to zero, but the cost grows rapidly. We will discuss improving it by replacing this memoryless coder with an automate having some small set of internal states: defined by "(symbol, state)→(bit sequence, new state)" set of rules. The natural question is the minimal number of states to achieve given performance, what is especially important for simple high throughput hardware coders. Arithmetic coding can be seen this way, but it requires relatively large number of states (possible ranges). We will discuss asymmetric numeral systems (ANS) for this purpose, which can be seen as asymmetrization of numeral systems. Less than 20 states will be usually sufficient to achieve $\Delta H \approx 0.001$ bits/symbol for a small alphabet. ΔH generally generally decreases approximately like 1/(the number of states)^2 and increases proportionally the size of alphabet. Huge freedom of choosing the exact coding and chaotic behavior of state make it also perfect to simultaneously encrypt the data.

1 Introduction

Electronics we use is usually based on the binary numeral system, which is perfect for handling integer number of bits of information. However, generally event/symbol of probability p contains $\lg(1/p)$ bits of information ($\lg \equiv \log_2$), which is not necessarily integer in real applications. If all probabilities would be integer powers of 3 instead, we could optimally use base 3 numeral system and analogously for larger bases. In more complex situations we need to use more sophisticated methods: entropy coders, translating between symbol sequence of some probability distribution and bit sequence. For fixed numbers of different symbols in the sequence, we can enumerate all possibilities (combinations) and encode given one by its number - this approach is called enumerative coding [1]. More practical are prefix codes, like Huffman coding [5], which is computationally

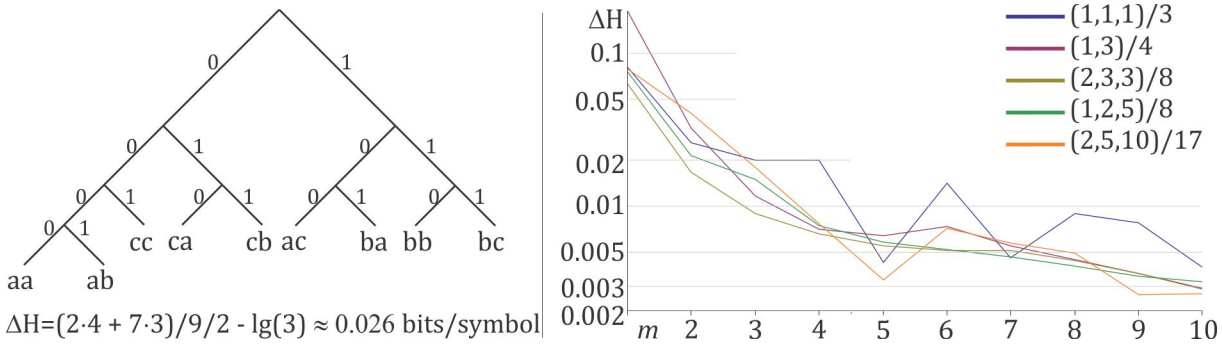

Figure 1: Left: construction of Huffman coding while grouping two symbols from $(1,1,1)/3$ probability distribution: we have 9 equiprobable possibilities. This grouping reduces distance from the Shannon entropy: ΔH. Standard base 3 numeral system would give $\Delta H = 0$ here. Right: ΔH for grouping m symbols. For example operating on $3^{10} = 59049$ possibilities allows to get loss $\Delta H \approx 0.003$ bits/symbol. In comparison, ΔH for ANS using just 16 or 17 states for the last four cases is correspondingly: ≈ 0.00065, 0.00122, 0.00147, 0.00121 bits/symbol.

inexpensive, but approximates probabilities with powers of 2, what reduces the capacity. We can improve it by grouping a few symbols together, but as we can see in Fig. 1, it is relatively expensive to get really close to the capacity (Shannon entropy). Precise analysis can be found in [9].

Encoding of a prefix code can be realized by memoryless automate using "symbol→bit sequence" set of rules. We will discuss improving its performance by adding some number of internal states to the automate, such that it can be defined for example by "(symbol, state)→(bit sequence, new state)" set of rules, which allow for unique decoding, like in Fig. 2. Intuitively, this state can be seen as a buffer storing noninteger number of bits ($\lg x \in [2,3)$ bits in this figure) - producing complete bits when they accumulate. Arithmetic/range coding ([6], [8]), which currently replaces Huffman coding due to better performance, can be seen this way. We usually use this coding by performing arithmetic calculations in every step (multiplication, division), which are more costly for hardware implementation than a few state automate with a fixed set of rules. We could put behavior of such coder working in a small discrete range into a lookup table, which could be used to construct an automate. The cost would be using rough approximation of probabilities.

Generally, if the we are using a coder which encodes perfectly (q_s) symbol distribution to encode (p_s) symbol sequence, we would use on average $\sum_s p_s \lg(1/q_s)$ bits per symbol, while there is only Shannon entropy needed: $\sum_s p_s \lg(1/p_s)$. The difference between them is called Kullback - Leiber distance:

$$\Delta H = \sum_s p_s \lg\left(\frac{p_s}{q_s}\right) \approx \sum_s \frac{-p_s}{\ln(2)}\left(\left(1-\frac{q_s}{p_s}\right) - \frac{1}{2}\left(1-\frac{q_s}{p_s}\right)^2\right) \approx 0.72 \sum_s \frac{(\epsilon_s)^2}{p_s} \quad (1)$$

where $\epsilon_s = q_s - p_s$ will be referred as *impreciseness*.

For range/arithmetic coding we approximate probabilities with proportions of lengths of subranges in succeeding steps. Restricting to a finite range, impreciseness of such

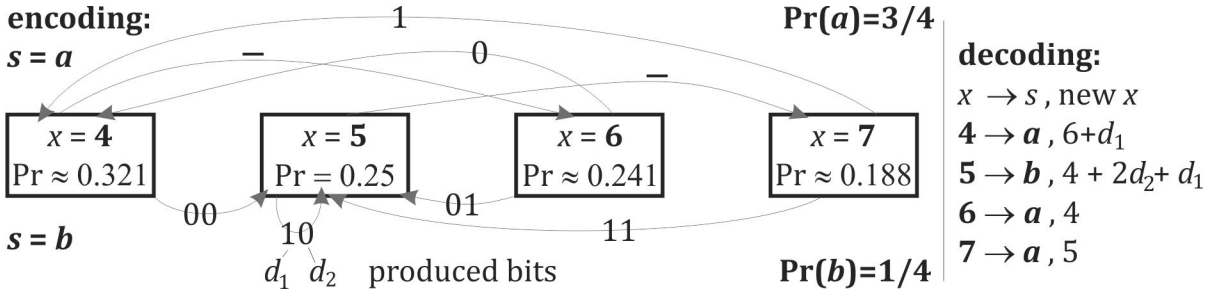

Figure 2: Some 4 state encoding and decoding automate for ANS with $\Pr(a) = 3/4$, $\Pr(b) = 1/4$ probability distribution. Upper edges of encoding picture are transitions for symbol "a", lower for symbol "b". Some edges contain digits produced while corresponding transition. Intuitively, x is a buffer containing $\lg(x) \in [2, 3)$ bits of information - symbol "b" always produces 2 bits, while "a" accumulates in the buffer. Decoding is unique because each state corresponds to a fixed symbol and number of digits to process: 1 for state 4, 2 for state 5, none for 6, 7. There is written stationary probability distribution for i.i.d. source, which allows to find average number of used bits per symbol: $\approx 2 \cdot 1/4 + 1 \cdot 3/4 \cdot (0.241 + 0.188) \approx 0.82$, what is larger than Shannon entropy by $\Delta H \approx 0.01$ bits/symbol. Increasing the number of states to 8 allows to reduce it to $\Delta H \approx 0.0018$ bits/symbol.

approximation should generally decrease inversely proportional to the size of this range. A state of such an automate would be a subrange, so the number of possible states would grow like square of size of the range we operate on. This number of states is relatively large and should grow like $1/\Delta H$ if we would like to get very close to the capacity.

The number of states would improve to $\propto 1/\sqrt{\Delta H}$ if the state would be a single number in the range instead of two (defining a subrange). While symbol of probability p contains $\lg(1/p)$ bits of information, we can imagine that a natural number (state) x contains $\lg(x)$ bits of information. So if we would like to store both information in a state $x' \in \mathbb{N}$, we should have $\lg(x') \approx \lg(x) + \lg(1/p) = \lg(x/p)$ and so $x' \approx x/p$. Observe that this relation is fulfilled in the binary numeral system, optimal for symmetric $\Pr(0) = \Pr(1) = 1/2$ probability distribution: while adding information from bit $s = 0, 1$ to the least significant position of x, we have $x \to 2x+s \approx x/\Pr(s)$. Recent asymmetric numeral systems (ANS) ([2], [3]) asymmetrizes this method to general distributions. Arithmetic coding can be also seen as such asymmetriztion, but for adding information in the most significant position - Fig. 3 compares these two approaches.

While using $x \to 2x + s$ formula to add information from s to x, we are choosing between x-th appearances of two subsets of $\mathbb{N}$: between even ($s = 0$) and odd numbers ($s = 1$). For a general distribution we need to redefine these subsets, such that they should still cover $\mathbb{N}$ in nearly uniform way, but have densities close to the assumed probability distribution. Now to add information from a given symbol to information stored in x, we will change x into x-th element of the subset corresponding to this symbol - it should have approximately x/p position. We will introduce the basic formalism and find analytic formulas for the binary case (ABS) in Section 2.

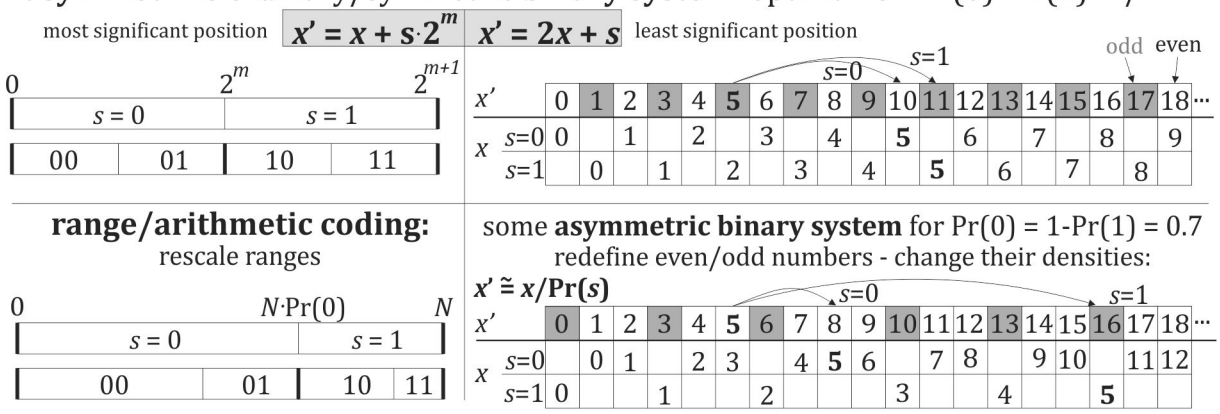

Figure 3: Two ways of asymmetrization of binary numeral system. Having some information stored in a natural number x, to attach information from 0/1 symbol s, we can add it in the most significant position ($x' = x + s2^m$), where s chooses between ranges, or in the least significant ($x' = 2x + s$) position, where s chooses between even and odd numbers. The former asymmetrizes to arithmetic/range coding by changing range proportions. The latter asymmetrizes to ABS by redefining even/odd numbers, such that they are still uniformly distributed, but with different density. Now x' is x-th element of the s-th subset.

While using pure arithmetic coding, we would need to operate with precision growing to infinity - to prevent that, we need some mechanism extracting bits of information as they accumulate, such that we remain in given precision (range of natural numbers in range coding). Here we also need this kind of mechanism - this time because x would grow to infinite. Section 3 discusses such stream version of ABS/ANS, enforcing x to stay in some chosen range $I = \{l, .., 2l - 1\}$. So before encoding succeeding symbol, we will first send some of the least significant bits of x to the data stream, such that encoding this symbol from such reduced state takes us back to I. While stream decoding, the current state defines the succeeding symbol, reducing the state - we should take some bits from the stream to take it back to I.

We can see I as the set of states of our encoding/decoding automate, like $I = \{4, 5, 6, 7\}$ in Fig. 2. To calculate the expected number of bits per symbol for a prefix code, we just sum probability of using given symbol times the number of bits it corresponds to, like in Fig. 1. While using automate with a state, it is a bit more complicated. First we need to find the stationary probability distribution of states it uses. We will assume i.i.d. input source here for analyzing given automate - the stationary probability is normalized dominant eigenvector of stochastic process of jumping on this graph. For example in Fig. 2, we can get to state 7 only if starting from state 5 and getting symbol a: $\Pr(7) = \Pr(5) \cdot \Pr(a)$. We will see that this distribution should be approximately $\Pr(x) \propto 1/x$. It is changed a bit if the source is correlated. We will find an upper bound for $\Delta H \propto 1/l^2$ which is independent from the exact distribution, and compare the formula with numerical results

A useful analytic formula is probably not known for larger than binary alphabet, but for given I we can directly generate the behavior to put into lookup tables - it will be discussed in Section 4. For this purpose we will distribute $l_s \approx lp_s$ appearances of symbol s on this range I, so we get the best behavior if lp_s are all natural: we should choose l to be a good denominator to approximate given distribution with fractions. The ΔH will approximately grow proportionally to the size of alphabet.

There is exponentially large number of ways to distribute symbols over I and each of them defines a concrete coder. We can use this freedom to choose the exact coder by a pseudorandom number generator initialized with a cryptographic key. This way, using larger number of states, we can simultaneously encrypt the data. We will discuss three reasons of chaotic behavior of internal state of encoder: asymmetry, ergodicity and diffusion, making tracing the state with incomplete knowledge extremely difficult.

There are available some implementations of ABS, for example of Matt Mahoney [7]. There is also available interactive demonstration of ANS based encoder [4].

2 Basic concepts and asymmetric binary systems

We will now introduce basic concepts for encoding information in natural numbers and find analytic formulas for the binary case: asymmetric binary systems (ABS).

2.1 Basic concepts

There is given an alphabet $\mathcal{A} = \{0, .., n-1\}$ (in some examples we will use letters instead) and assumed probability distribution $\{p_s\}_{s\in\mathcal{A}}$, $\sum_s p_s = 1$. The *state* of automate, denoted as $x \in \mathbb{N}$, in this section will contain already processed symbol sequence.

We need to find *encoding* (C) and *decoding* (D) *functions*. The former takes the state $x \in \mathbb{N}$ and a symbol $s \in \mathcal{A}$, and transforms them into $x' \in \mathbb{N}$ storing information from both of them. Seeing x as a possibility of choosing a number from $\{0, 1, .., x-1\}$ interval, it contains $\lg(x)$ bits of information. Symbol of probability p_s contains $\lg(1/p_s)$ bits of information, so x' should contain approximately $\lg(x) + \lg(1/p_s) = \lg(x/p_s)$ bits of information: x' should be approximately x/p_s, allowing to choose a value from a larger interval $\{0, 1, .., x'-1\}$. Finally we will have functions defining single steps:

$$C(s,x) = x', \quad D(x') = (s,x) \quad : \quad D(C(s,x)) = (s,x), \quad C(D(x')) = x', \quad x' \approx x/p_s$$

For standard binary system we have $C(s,x) = 2x + s$, $D(x') = (x - 2\lfloor x/2\rfloor, \lfloor x/2\rfloor)$ and we can see s as choosing between even and odd number. If we imagine that x chooses between even (or odd) numbers in some interval, $x' = 2x + s$ chooses between all numbers in this interval.

For the general case, we will have to redefine the subsets corresponding to different s: like even/odd numbers they should still uniformly cover $\mathbb{N}$, but this time with different densities: $\{p_s\}_{s\in\mathcal{A}}$. We can define this split of $\mathbb{N}$ by a *symbol distribution* $\overline{s} : \mathbb{N} \to \mathcal{A}$

$$\{0, 1, .., x'-1\} = \bigcup_s \{x \in \{0, 1, .., x'-1\} : \overline{s}(x) = s\}$$

While in standard binary system x' is x-th appearance of even/odd number, this time it will be x-th appearance of s-th subset. So the decoding function will be

$$D(x) = (\overline{s}(x), x_{\overline{s}(x)}) \qquad \text{where} \qquad x_s := |\{y \in \{0,1,..,x-1\} : \overline{s}(y) = s\}| \tag{2}$$

and $C(s, x_s) = x$ is its inversion. Obviously we have $x = \sum_s x_s$.

As x/x_s is the number of bits we currently use to encode symbol s, to reduce impreciseness and so ΔH, we would like that $x_s \approx xp_s$ approximation is as close as possible - what intuitively means that symbols are nearly uniformly distributed with $\{p_s\}$ density. We will now find formulas for the binary case by just taking $x_1 := \lceil xp \rceil$ and in Section 4 we will focus on finding such nearly uniform distributions on a fixed interval for larger alphabets.

We can imagine that x is a stack of symbols, C is push operation, D is pop operation. The current encoding algorithm would be: start with e.g. $x = 1$ and then use C with succeeding symbols. It would lead to a large natural number, from which we can extract all the symbols in reversed order using D. To prevent inconvenient operations on large numbers, in the next section we will discuss stream version, in which complete cumulated bits will be extracted to make that x remains in a fixed interval.

2.2 Asymmetric binary systems (ABS)

We will now find some explicit formulas for the binary case: $\mathcal{A} = \{0, 1\}$.

Denote $p := p_1$, $\tilde{p} := 1 - p = p_0$. To obtain $x_s \approx x \cdot p_s$ we can for example choose

$$x_1 := \lceil xp \rceil \qquad \text{(or alternatively } x_1 := \lfloor xp \rfloor) \tag{3}$$

$$x_0 = x - x_1 = x - \lceil xp \rceil \qquad \text{(or } \; x_0 = x - \lfloor xp \rfloor) \tag{4}$$

Now $\overline{s}(x) = 1$ if there is a jump of $\lceil xq \rceil$ in the succeeding position:

$$s := \lceil (x+1)p \rceil - \lceil xp \rceil \qquad \text{(or } s := \lfloor (x+1)p \rfloor - \lfloor xq \rfloor) \tag{5}$$

This way we have found some **decoding** function: $D(x) = (s, x_s)$.

We will now find the corresponding encoding function: for given s and x_s we want to find x. Denote $r := \lceil xq \rceil - xq \in [0, 1)$

$$\overline{s}(x) = s = \lceil (x+1)p \rceil - \lceil xp \rceil = \lceil (x+1)q - \lceil xq \rceil \rceil = \lceil (x+1)p - r - xq \rceil = \lceil p - r \rceil \tag{6}$$

$$s = 1 \Leftrightarrow r < p$$

- $s = 1$: $\quad x_1 = \lceil xp \rceil = xp + r$
  $x = \frac{x_1 - r}{p} = \left\lfloor \frac{x_1}{p} \right\rfloor$ as it is a natural number and $0 \le r < p$.
- $s = 0$: $p \le r < 1$ so $\tilde{p} \ge 1 - r > 0$
  $x_0 = x - \lceil xp \rceil = x - xp - r = x\tilde{p} - r$

$$x = \frac{x_0 + r}{\tilde{p}} = \frac{x_0 + 1}{\tilde{p}} - \frac{1 - r}{\tilde{p}} = \left\lceil \frac{x_0 + 1}{\tilde{p}} \right\rceil - 1$$

Finally **encoding** is:

$$C(s,x) = \begin{cases} \left\lceil \frac{x+1}{1-p} \right\rceil - 1 & \text{if } s=0 \\ \left\lfloor \frac{x}{p} \right\rfloor & \text{if } s=1 \end{cases} \qquad \left(\text{or } = \begin{cases} \left\lfloor \frac{x}{1-p} \right\rfloor & \text{if } s=0 \\ \left\lceil \frac{x+1}{p} \right\rceil - 1 & \text{if } s=1 \end{cases} \right) \tag{7}$$

For $p = 1/2$ it is the standard binary numeral system with switched digits.

The starting values for this formula and $p = 0.3$ are presented in bottom-right of Fig. 3. Here is an example of encoding process by inserting succeeding symbols:

$$1 \xrightarrow{1} 3 \xrightarrow{0} 5 \xrightarrow{0} 8 \xrightarrow{1} 26 \xrightarrow{0} 38 \xrightarrow{1} 128 \xrightarrow{0} 184 \xrightarrow{0} 264... \tag{8}$$

We could directly encode the final x using $\lfloor \lg(x) \rfloor + 1$ bits. Let us look at the growth of $\lg(x)$ while encoding: symbol s transforms state from x_s to x:

$$-\lg\left(\frac{x_s}{x}\right) = -\lg\left(p_s + \epsilon_s(x)\right) = -\lg(p_s) - \frac{\epsilon_s(x)}{p_s \ln(2)} + O((\epsilon_s(x))^2)$$

where

$$\epsilon_s(x) = x_s/x - p_s \qquad \text{describes impreciseness.}$$

In the found coding we have $|x_s - x p_s| < 1$ and so $|\epsilon_s(x)| < 1/x$. While encoding symbol sequence of $\{p_s\}_{s\in\mathcal{A}}$ probability distribution, the sum of above expansion says that we need on average $H = -\sum_s p_s \lg(p_s)$ bits/symbol plus higher order terms. As x grows exponentially while encoding, $|\epsilon_s(x)| < 1/x$, so these corrections are $O(1)$ for the whole sequence.

3 Stream version - encoding finite-state automate

Using arithmetic coding to directly encode a long sequence would require growing precision of arithmetics and so cost increasing to infinity. ABS has the same issue, as x would grow exponentially. To handle this issue, we will enforce x to remain in some chosen range $I = \{l, .., 2l-1\}$, so x can be seen as a buffer containing between l and $l+1$ bits of information. While we operate on symbols containing non-integer number of bits, as they accumulate, we can extract complete bits of information to remain in the buffer size, and send them to the data stream. Finally the situation will intuitively look like in Fig. 4.

3.1 Algorithm

To make these considerations more general, we will extract digits in base $2 \leq b \in \mathbb{N}$ numeral system. Usually we will use bits: $b = 2$, but sometimes using a larger b could be more convenient, for example $b = 2^k$ allows to extract k bits at once. Extracting the least significant base b digit means: $x \to \lfloor x/b \rfloor$ and $x - b\lfloor x/b \rfloor$ goes to the stream.

Observe that taking interval in form ($l \in \mathbb{N}$):

$$I := \{l, l+1, .., lb-1\} \tag{9}$$

for any $x \in \mathbb{N}$ we have exactly one of three cases:

- $x \in I$ or
- $x > lb-1$, then $\exists!_{k\in\mathbb{N}}\ \lfloor x/b^k \rfloor \in I$ or
- $x < l$, then $\forall_{(d_i)_i\in\{0,..,b-1\}^{\mathbb{N}}}\ \exists!_{k\in\mathbb{N}}\ xb^k + d_1 b^{k-1} + .. + d_k \in I$.

We will refer to this kind of interval as *b-unique* as eventually inserting ($x \to bx+d$) or removing ($x \to \lfloor x/b \rfloor$) some of the the least significant digits of x, we will always finally get to I in a unique way. Let us also define

$$I_s = \{x : C(s,x) \in I\} \qquad \text{so} \qquad I = \bigcup_s C(s, I_s) \tag{10}$$

We can now define single steps for stream decoding: use D function and then get the least significant digits from the steam until we get back to I, and encoding: send the least significant digits to the stream, until we can use the C function:

Stream decoding:
$(s,x) = D(x)$;
`use` s; (e.g. to generate symbol)
`while(`$x \notin I$`)`
$x = xb+$`'digit from input'`
`}`

Stream encoding(`s`):
`{while(`$x \notin I_s$`)`
`{put mod(`x,b`) to output;` $x = \lfloor x/b \rfloor$`}`
$x = C(s,x)$
`}`

These functions are inverse of each other if only I_s are also b-unique: there exists $\{l_s\}_{s\in\mathcal{A}}$ such that

$$I_s = \{l_s, ..., l_s b - 1\} \tag{11}$$

We need to ensure that this necessary condition is fulfilled, what is not automatically true as we will see in the example. In Section 3.3 we find its more compact form for the ABS formulas and for larger alphabets we will directly enforce its fulfillment in Section 4.

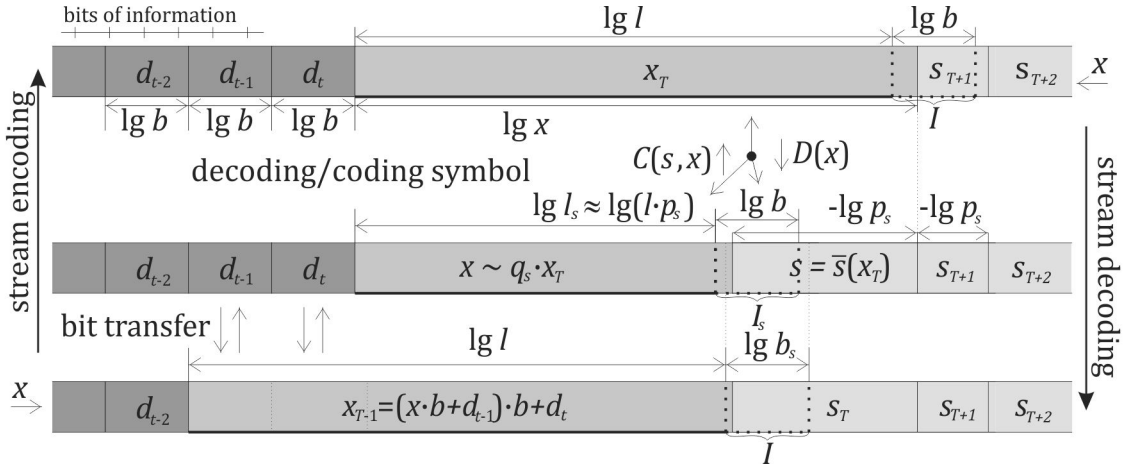

Figure 4: Schematic picture of stream encoding/decoding: encoder travels right, transforming symbols containing $\lg(1/p_s)$ bits of information, into digits containing $\lg(d)$ bits each. When information accumulates in buffer x, digits are produced to make that x remain in $I = l, .., bl-1$. Decoder analogously travels left.

3.2 Example

Taking ABS for $p = 0.3$ as in Fig. 3, let us transform it to stream version for $b = 2$ (we transfer single least significant bits). Choosing $l = 8$, we have $x \in I = \{8, .., 15\}$. We can see from this figure that $I_0 = \{5, .., 10\}$, $I_1 = \{3, 4\}$ are the ranges from which we get to I after encoding corresponding 0 or 1. These ranges are not b-unique, so for example if we would like to encode $s = 1$ from $x = 10$, while transferring the least significant bits we would first reduce it to $x = 5$ and then to $x = 2$, not getting into the I_1 range.

However, if we choose $l = 9$, $I = \{9, .., 17\}$, we get $I_0 = \{6, .., 11\}$ and $I_1 = \{3, 4, 5\}$ which are b-unique. Here is the encoding process for $s = 0, 1$: first we transfer some of the least significant bits to the stream, until we reduce x to I_s range. Then we use ABS formula (like in Fig. 3):

$x \in I$	9	10	11	12	13	14	15	16	17
bit transfer for $s = 0$	-	-	-	0	1	0	1	0	1
reduced $x' \in I_0$	9	10	11	6	6	7	7	8	8
$\overline{C}(0, x) = C(0, x')$	14	15	17	9	9	11	11	12	12
bit transfer for $s = 1$	1	0	1	0, 0	1, 0	0, 1	1, 1	0, 0	1, 0
reduced $x' \in I_1$	4	5	5	3	3	3	3	4	4
$\overline{C}(1, x) = C(1, x')$	13	16	16	10	10	10	10	13	13

Here is an example of evolution of (state, bit sequence) while using this table:

$$(9, -) \xrightarrow{1} (13, 1) \xrightarrow{0} (9, 11) \xrightarrow{0} (14, 11) \xrightarrow{1} (10, 1101) \xrightarrow{0} (15, 1101) \xrightarrow{1} (10, 110111)...$$

The decoder will first use $D(x)$ to get symbol and reduced x', then add the least significant bits from stream (in reversed order).

To find the expected number of bits/symbol used by such process, let us first find its stationary probability distribution assuming i.i.d. input source. Denoting by $\overline{C}(s, x)$ the state to which we go from state x due to symbol s (like in the table above), this stationary probability distribution have to fulfill:

$$\Pr(x) = \sum_{s,y:\overline{C}(s,y)=x} \Pr(y) p_s \tag{12}$$

equation, being the dominant eigenvector of corresponding stochastic matrix. Such numerically found distribution is written in Fig. 2. Here is approximated distribution for currently considered automate and comparison with close $\Pr(x) \propto 1/x$ distribution, what will be motivated in Section 3.5:

x	9	10	11	12	13	14	15	16	17
$\Pr(x)$	0.1534	0.1240	0.1360	0.1212	0.0980	0.1074	0.0868	0.0780	0.0952
$1.3856/x$	0.1540	0.1386	0.1260	0.1155	0.1066	0.0990	0.0924	0.0866	0.0815

We can now find the expected number of bits/symbol used by this automate by summing the number of bits used in encoding table:

$$\left(\sum_{x=12}^{17} \Pr(x)\right) p_0 + \left(\Pr(9) + \Pr(10) + \Pr(11) + 2\sum_{x=12}^{17} \Pr(x)\right) p_1 \approx 0.88658 \text{ bits/symbol}$$

For comparison, Shannon entropy is $h(p) = -p\lg(p) - (1-p)\lg(1-p) \approx 0.88129$ bits/symbol. So this automate uses about $\Delta H \approx 0.00529$ bits/symbol more than required.

The found encoding table fully determines the encoding process and analogously we can generate the table for decoding process, defining automate like in Fig. 2.

Observe that having a few such encoders operating on the same range I, we can use them consecutively in some order - if decoder will use them in reversed order, it will still retrieve the encoded message. Such combination of multiple different encoder can be useful for example when probability distribution varies, or for cryptographic purposes.

3.3 Necessary condition and remarks for stream ABS

For unique decoding we need that all I_s are b-unique, what is not always true as we have seen in the example above. In the next section we will enforce it by construction for larger alphabet. We will now check when the ABS formula fulfills this condition.

In the binary case we need to ensure that both I_0 and I_1 are b-absorbing: denoting $I_s = \{l_s, .., u_s\}$, we need to check that

$$u_s = bl_s - 1 \qquad \text{for } s = 0, 1$$

We have $l_s = |\{x < l : \overline{s}(x) = s\}|$, $u_s = |\{x < bl : \overline{s}(x) = s\}| - 1$, so $\sum_s l_s = l$, $\sum_s u_s = bl - 2$. It means that fulfilling one of these condition implies the second one.

Let us check when $u_1 = bl_1 - 1$. We have $l_1 = \lceil lp \rceil$, $u_1 = \lceil blp \rceil - 1$, so the condition is:
Condition: Stream ABS can be used when

$$b\lceil lp \rceil = \lceil blp \rceil. \tag{13}$$

The basic situation this condition is fulfilled is when $lp \in \mathbb{N}$, what means that p is defined with $1/l$ precision.

While using ABS there are two basic possibilities ([7] implementation fpaqa, fpaqc):

- use the formulas directly for example in 32 bit arithmetics (fpaqc) - as $l > 2^{20}$, impreciseness becomes negligible, we can use large b to extract a few bits at once. However, the arithmetic operation can make it a bit slower,
- store behavior on some chosen range (fpaqa) - it is less precise, but can be a bit faster and leaves freedom to choose exact encoding - we will explore this possibility with ANS.

If probability varies, in the former case we just use the formulas for the current p, while in the latter case we should have prepared tables for differnt probability distributions we could use for approximation. Such varying of p is allowed as long l and b remain fixed.

3.4 Analysis of a single step

Let us now look closer at a single step of stream encoding and decoding. Observe that the number of digits to transfer to get from x to $I_s = \{l_s, .., bl_s - 1\}$ is $k = \lfloor \log_b(x/l_s) \rfloor$, so the (new symbol, digit sequence) while stream coding is:

$$x \xrightarrow{s} \left(C(s, \lfloor x/b^k \rfloor),\ x - b\lfloor x/b^k \rfloor\right) \qquad \text{where} \quad k = \lfloor \log_b(x/l_s) \rfloor \tag{14}$$

this $x - b\lfloor x/b^k \rfloor$ is the information lost from x while digit transfer - this value is being sent to the stream in base d numeral system. The original algorithm produces these digits in reversed order: from less to more significant.

Observe that for given symbol s, the number of digits to transfer can obtain one of two values: $k_s - 1$ or k_s for $k_s := \lfloor \log_b(x/l_s) \rfloor + 1$. The smallest x requiring to transfer k_s digits is $X_s := lb^{k_s}$. So finally the number of bits to transfer while encoding is $k_s - 1$ for $x \in \{l, .., X_s - 1\}$ and k_s for $x \in \{X_s, .., lb - 1\}$, like in Fig. 5.

While stream decoding, the current state x defines currently produced symbol s. In example in Fig. 2, the state also defined the number of digits to take: 1 for $x = 4$, 2 for $x = 5$ and 0 for $x = 6, 7$.

However, in example from this section, state $x = 13$ is an exception: it is decoded to $s = 1$ and $x' = 4$. Now if the first digit is 1, the state became $x = 2 \cdot 4 + 1 = 9 \in I$. But if the first digit is 0, it became $2 \cdot 4 = 8 \notin I$ and so we need to take another digit, finally getting to $x = 16$ or 17. We can also see such situation in Fig. 5.

This issue - that decoding from a given state may require $k_s - 1$ or k_s digits to transfer, means that for some $x \in \mathbb{N}$ we have: $bx < l$, while $bx + b - 1 \geq l$. It is possible only if b does not divide l.

To summarize: if b divides l, decoding requires to transfer a number of digits which is fixed for every state x. So we can treat these digits $(x - b\lfloor x/b^k \rfloor)$ as blocks and store them in forward or backward order. However, if b does not divide l, sometimes state alone does not determine the number of digits: we should first add $k_s - 1$ first digits, and if we are still not in I, add one more. We see that in this case, it is essential that digits are in the

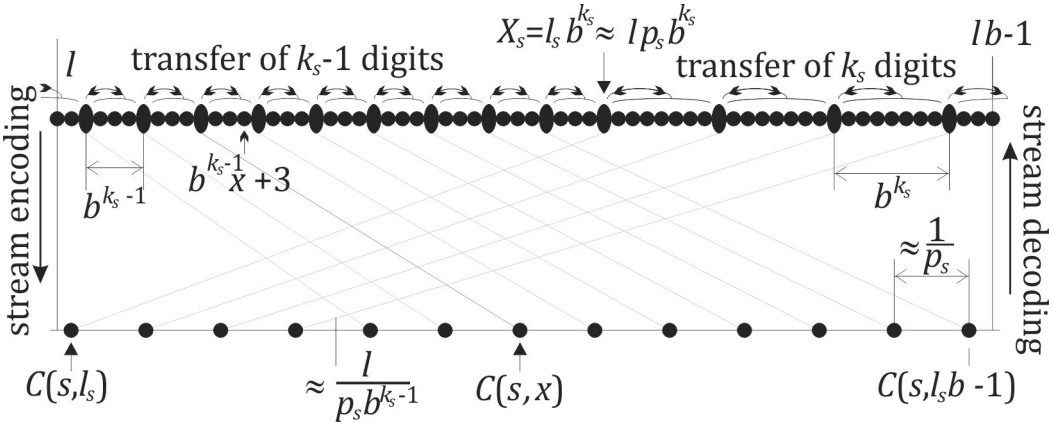

Figure 5: Example of stream coding/decoding step for $b = 2$, $k_s = 3$, $l_s = 13$, $l = 9 \cdot 4 + 3 \cdot 8 + 6 = 66$, $p_s = 13/66$, $x = 19$, $b^{k_s - 1}x + 3 = 79 = 66 + 2 + 2 \cdot 4 + 3$.

proper order (reversed): from the least to the most significant.

3.5 Stationary probability distribution of states

In Section 3.2 there was mentioned finding the stationary probability distribution of states used while encoding (and so also while decoding) - fulfilling (12) condition. We will now motivate that this distribution is approximately $\Pr(x) \propto 1/x$.

Let us transform I range into $[0, 1)$:

$$y = \log_b(x/l) \in [0, 1) \tag{15}$$

Encoding symbol of probability p_s increases y by approximately $\log_b(1/p_s)$. However, to remain in the range, sometimes we need to transfer some number of digits before. Each digit transfer reduces x to $\lfloor x/d \rfloor$ and so y by approximately 1:

$$y \xrightarrow{s} \approx \{y + \log_b(1/p_s)\}$$

where $\{a\} = a - \lfloor a \rfloor$ denotes the fractional part.

While neighboring values in arithmetic coding remain close to each other when adding succeeding symbols (ranges are further compressed), here we have much more chaotic behavior, making it more appropriate for cryptographic applications. There are three sources of its chaosity, visualized in Fig. 6:

- *asymmetry*: different symbols can have different probability and so different shift,
- *ergodicity*: $\log_b(1/p_s)$ is usually irrational, so even a single symbol should lead to uniform covering of the range,
- *diffusion*: $\log_b(1/p_s)$ only approximates the exact shift and this approximation varies with position - leading to pseudorandom diffusion around the expected position.

These reasons strongly suggest that probability distribution while encoding is nearly uniform for y variable: $\Pr(y \leq a) \approx a \in [0, 1]$, what is confirmed by numerical results. Transforming it back to x variable, probability of using $x \in I$ state is approximately proportional to $1/x$

$$\Pr(x) \propto 1/x \tag{16}$$

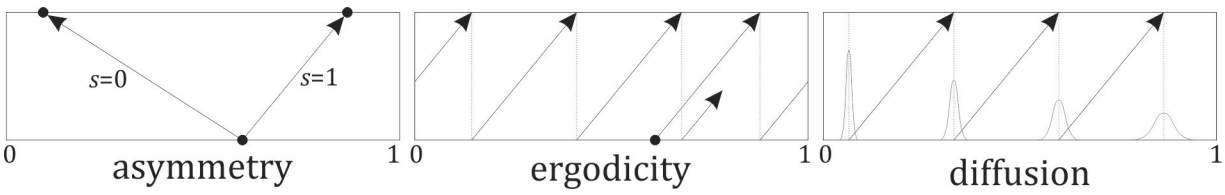

Figure 6: Three sources of chaotic behavior of $y = \log_b(x/l)$, leading to nearly uniform distribution on the $[0, 1]$ range.

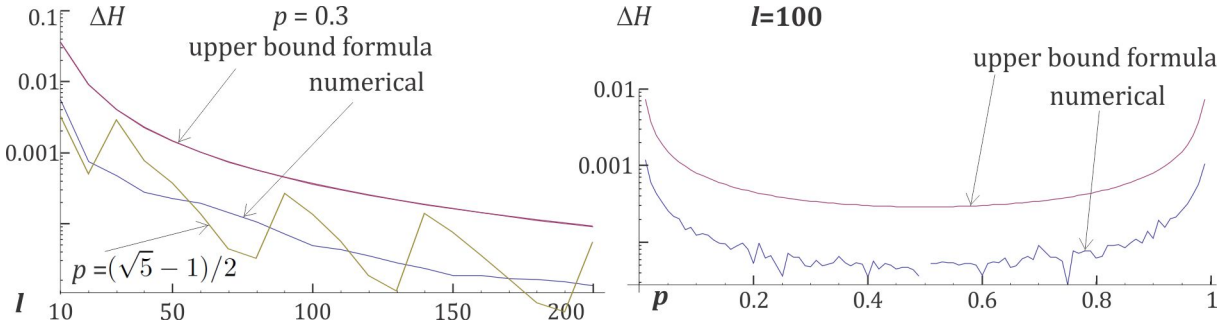

Figure 7: Left: ΔH for $p = 0.3$ and different l being multiplicities of 10. The red line is formula (17). Additionally, the irregular yellow line is situation in these l when we approximate some irrational number using fractions with denominator l. Right: situation for fixed $l = 100$ and $p = i/100$ for $i = 1, 2, .., 99$. Discontinuity for $i = 50$ is because $\Delta H = 0$ there.

3.6 Bound for ΔH

We will now find some upper bound for the distance from Shannon entropy ΔH. As finding the exact stationary probability is a complex task and this distribution can be disturbed by eventual correlations, we will use the fact that impreciseness $|\epsilon_s(x)| \leq 1/x$ for ABS to find a bound which is independent from this distribution.

Generally, assuming that the probability distribution is indeed $\{p_s\}$, but we pay $\lg(1/q_s)$ bits per symbol s, the cost of impreciseness is Kullback-Leiber distance:

$$\Delta H = \sum_s p_s \lg(1/q_s) - \sum_s p_s \lg(1/p_s) = -\sum_s p_s \lg\left(1 - \left(1 - \frac{q_s}{p_s}\right)\right) =$$

$$= \sum_s \frac{p_s}{\ln(2)} \left(\left(1 - \frac{q_s}{p_s}\right) + \frac{1}{2}\left(1 - \frac{q_s}{p_s}\right)^2 + O\left(\left(1 - \frac{q_s}{p_s}\right)^3\right)\right) \approx \sum_s \frac{(p_s - q_s)^2}{p_s \ln(4)}$$

In our case we use $\lg(x/x_s)$ bits to encode symbol s from state x_s, so $(p_s - q_s)^2$ corresponds to $|\epsilon_s(x)|^2 < 1/l^2$ for $x \in I = \{l, .., lb - 1\}$. The above sum should also average over the encoder state, but as we are using a general upper bound for ϵ, we can bound the expected capacity loss of ABS on $I = \{l, .., bl - 1\}$ range to at most:

$$\Delta H \leq \frac{1}{l^2 \ln(4)} \sum_{s=0,1} \frac{1}{p_s} + O(l^{-3}) \text{ bits/symbol} \tag{17}$$

From Fig. 7 we see that it is a rough bound, but it reflects well the general behavior.

3.7 Direction of encoding/decoding

Encoding and decoding in discussed method are in reversed direction - which is perfect if we need a stack for symbols, but generally may be an inconvenience. One issue is the need of storing the final state of encoding, which will be required to start decoding - the cost is a few bits of information for every frame of data, which length is not bounded.

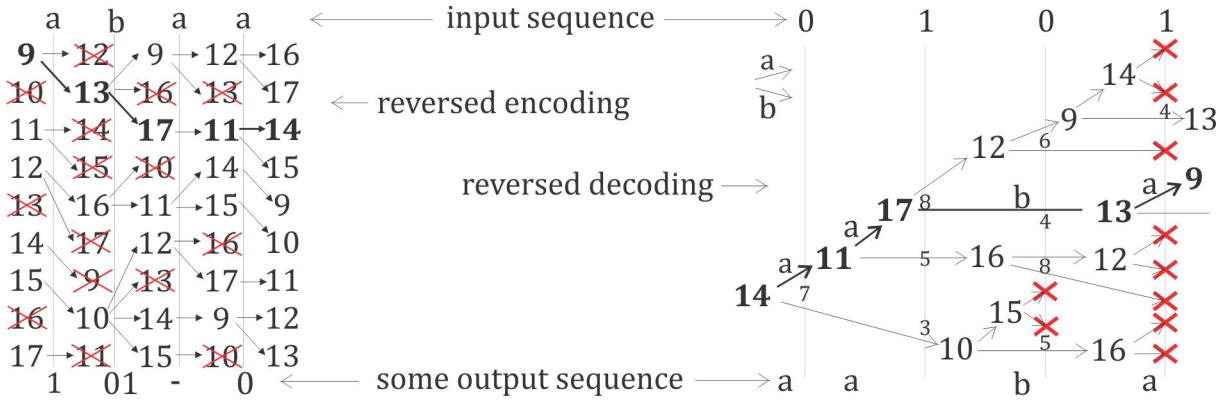

Figure 8: Example of encoding (left) and decoding (right) in reversed direction for encoder from Section 3.2 - it is much more costly, but still possible. We find some output sequence, such that later correspondingly decoding or encoding in the same direction will produce the input sequence. While reversed encoding we can start with all possible states, then in every step remove those not corresponding to given symbol and make a step for every possible digits. While reversed decoding, here from a single state, we try to encode all possible symbols from given state (arrow up: a, arrow down: b) and check if the input bits agree with the least significant bits. The written small numbers are after removing this least significant bit. Finally, in both cases we choose a single path. There is always such a path as we could perform this encoding/decoding in standard direction starting with any state.

Another issue is when probabilities depend on already processed data - in this case we can encode the data in backward direction, using information available while later decoding in forward direction. For example for Markov source of $s_1..s_m$ symbols, we would encode in backward direction: from m-th to the 1-st, but the probability of s_k would depend on s_{k-1}. For adaptive encoding, we would need to process the whole data frame in forward direction, assigning to each position probability used while decoding, then encode in backward direction using theses probabilities. Thanks of that, we can later use standard Markov or adaptive forward decoding.

However, if it is really necessary to encode and decode in the same direction, it is possible to change direction of encoding or decoding, but it is much more costly. We can see example of a few steps of such process in Fig. 8.

It is an interesting research question to find low state entropy coders which can directly encode and decode in the same direction, maybe as a subfamily of presented method, or some reduction of arithmetic coding approach, or maybe some completely new approach.

4 Asymmetric numeral systems (ANS)

While practical formulas for binary alphabet were derived in Section 2.2, for larger alphabet situation seems to be more complex. However, we can directly generate tables used to encoding/decoding, like in example from Section 3.2. Generally, large alphabets can be

processed by splitting it into a few binary choices - the advantage of directly using large alphabet, for example obtained by grouping a few symbols, is both speed and simplicity: we can process many bits in a single cycle using simple automate. The cost is that ΔH grows in approximately linear way with the size of alphabet.

There is a large freedom of generating such table fulfilling approximate relation $x_s/x \approx p_s$. We will now focus on doing it in a very precise way to get very close the Shannon entropy. However, this process can be for example disturbed in pseudorandom way using a cryptographic key to additionally encrypt the message.

4.1 Algorithm

Assume we have given l, b and probability distribution of n symbols: $0 < p_1, .., p_n < 1$, $\sum_s p_s = 1$. For simplicity, let us assume that probabilities are defined with $1/l$ precision:

$$l_s := lp_s \in \mathbb{N} \tag{18}$$

The encoding is defined by distributing symbols in nearly uniform way on the $I = \{l, .., bl - 1\}$ range: $(b-1)l_s$ appearances of symbol s. Then for every symbol s we enumerate its appearances by succeeding numbers from $I_s = \{l_s, .., bl_s - 1\}$ range, like in Fig. 9.

Unfortunately, finding the optimal symbol distribution is not an easy task. We can do it by checking all symbol distributions and finding ΔH for each of them - for example for $l_1 = 10$, $l_2 = 5$, $l_3 = 2$ there are $\binom{17}{2,5} = 408408$ ways to distribute the symbols and each of them corresponds to a differed coding. Surprisingly, while checking all these possibilities it turns out that the minimal value of ΔH among them: $\Delta H \approx 0.00121$ bits/symbol, is obtained by 32 different distributions - they are presented in the right hand side part of Fig. 9. They can be obtained by switching pairs of neighboring nodes on some 5 positions (marked with thick lines).

We will now introduce and analyze a heuristic algorithm which directly chooses a symbol distribution in nearly uniform way. Example of its application is presented in Fig. 9. In this case it finds one of the optimal distributions, but it is not always true - sometimes there are a bit better distributions than the generated one. So if the capacity is a top priority while designing a coder, it can be reasonable to check all symbol distributions and generally we should search for a better distributing algorithm.

To formulate the heuristic algorithm, let us first define:

$$N_s := \left\{ \frac{1}{2p_s} + \frac{i}{p_s} : \ i \in \mathbb{N} \right\} \tag{19}$$

These n sets are uniformly distributed with required densities, but they get out of natural numbers - to define ANS we need to shift them there, like in Fig. 9. Specifically, to choose a symbol for the succeeding position, we can take the smallest not used element from $N_1, .., N_n$ sets. This simple algorithm requires a priority queue to retrieve the smallest of currently considered n values. Beside initialization, this queue needs

two instructions: let `put`$((v,s))$ insert pair (v,s) with value $v \in N_s$ pointing expected succeeding position for symbol s. The second instruction: `getmin` removes and returns pair which is the smallest for $(v,s) \leq (v',s') \Leftrightarrow v \leq v'$ relation. Finally the algorithm is:

Precise initialization of encoding or decoding function:
`For` $s=0$ `to` $n-1$ `do {put(` $(0.5/p_s, s)$ `);` $x_s = l_s$`};`
`For` $x = l$ `to` $bl-1$ `do`
`{`(v,s)`=getmin; put(`$(v+1/p_s, s)$`);`
$D[x] = (s, x_s)$ or $C[s, x_s] = x$
x_s`++}`

There has remained a question which symbol should be chosen if two of them have the same position. Experiments suggest to choose the least probable symbol among them first, like in example in Fig. 9 (it can be realized for example by adding some small values to N_s). However, there can be found examples when opposite approach: the most probable first, brings a bit smaller ΔH.

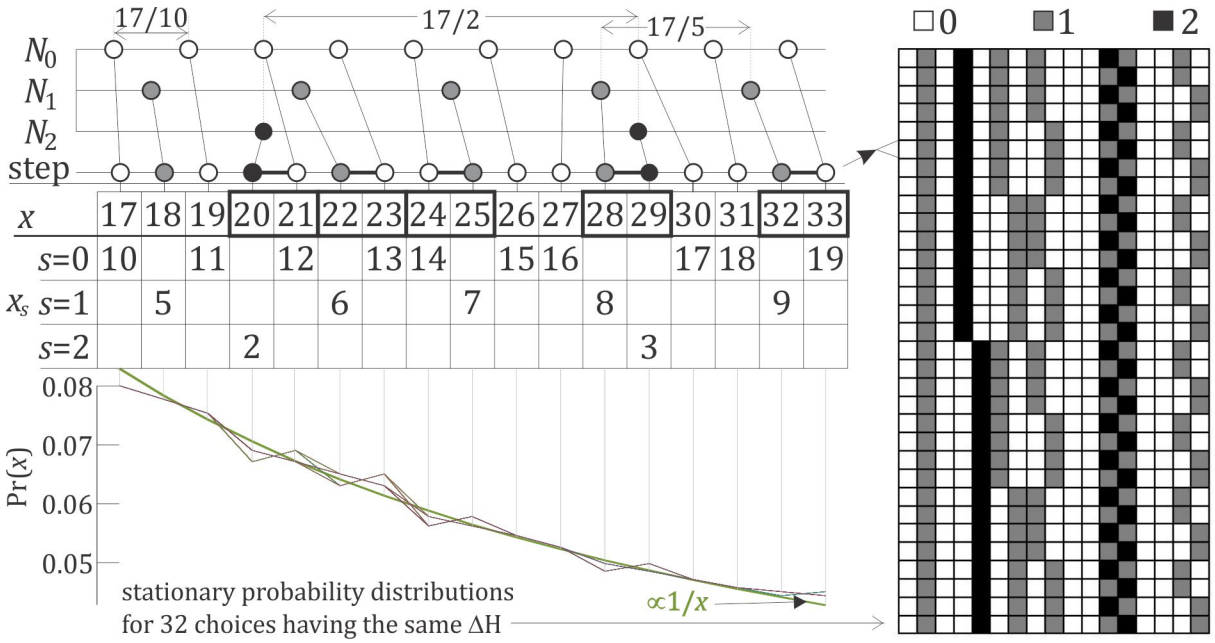

Figure 9: Left from top: precise algorithm initialization for $(10,5,2)/17$ probability distribution, corresponding encoding table and stationary probability distributions. Right: among $\binom{17}{2,5}$ possibilities to choose symbol distribution here, the minimal ΔH turns out to be obtained by 32 of them - presented in graphical form. The sixth from the top is the one generated by our algorithm. These 32 possibilities turn out to differ by switching two neighboring positions in 5 locations ($2^5 = 32$) - these pairs are marked by thick lines. We can see that these 5 positions correspond to the largest deviations from $\propto 1/x$ expected behavior of stationary probability distribution.

4.2 Impreciseness bound

Let us now find some bound for impreciseness ϵ for this algorithm to get an upper bound for ΔH. Observe that the symbol sequence it produces has period l, so for simplicity we us can imagine that it starts with $x = 0$ instead of $x = l$.

From definition, $\#(N_s \cap [0, x - 1/(2p_s)]) = \lfloor xp_s \rfloor$. As $\lfloor xp_s \rfloor \leq xp_s \leq \lfloor xp_s + 1 \rfloor$ and $\sum_s p_s = 1$, we have also $\sum_s \lfloor xp_s \rfloor \leq x \leq \sum_s \lfloor xp_s + 1 \rfloor$. Defining $\overline{p} = \min_s p_s$, we can check that in $[0, x - 1/(2\overline{p})]$ range there is at most x symbols in all N_s and in $[0, x + 1/(2\overline{p})]$ range there is at least x symbols:

$$\sum_s \#\left(N_s \cap \left[0, x - \frac{1}{2\overline{p}}\right]\right) \leq \sum_s \#\left(N_s \cap \left[0, x - \frac{1}{2p_s}\right]\right) = \sum_s \lfloor xp_s \rfloor \leq x$$

$$x \leq \sum_s \lfloor xp_s + 1 \rfloor = \sum_s \#\left(N_s \cap \left[0, x + \frac{1}{2p_s}\right]\right) \leq \sum_s \#\left(N_s \cap \left[0, x + \frac{1}{2\overline{p}}\right]\right)$$

So x-th symbol found by the algorithm had to be chosen from $[x - 1/(2\overline{p}), x + 1/(2\overline{p})] \cap \bigcup_s N_s$. From definition, the x_s-th value of N_s is $1/(2p_s) + x_s/p_s$. If this value corresponds to x, it had to be in the $[x - 1/(2\overline{p}), x + 1/(2\overline{p})]$ range:

$$\left|\frac{1}{2p_s} + \frac{x_s}{p_s} - x\right| \leq \frac{1}{2\overline{p}} \qquad \Rightarrow \qquad \left|\frac{1}{2x} + \frac{x_s}{x} - p_s\right| \leq \frac{p_s}{2x\overline{p}}$$

$$|\epsilon_s(x)| = \left|\frac{x_s}{x} - p_s\right| \leq \left(\frac{p_s}{2\overline{p}} + \frac{1}{2}\right)/x \tag{20}$$

Finally in analogy to (17) we get:

$$\Delta H \leq \frac{1}{l^2 \ln(4)} \sum_s \frac{1}{p_s} \left(\frac{p_s}{2 \min_{s'} p_{s'}} + \frac{1}{2}\right)^2 + O(l^{-3}) \text{ bits/symbol} \tag{21}$$

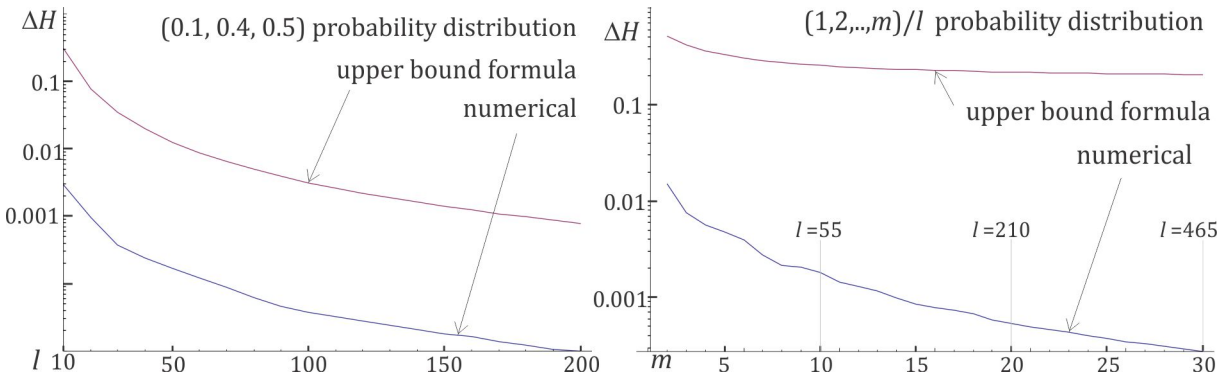

Figure 10: Left: numerical values and comparison with the (21) formula for $(0.1, 0.4, 0.5)$ probability distribution and l being a multiplicity of 10. Right: comparison for larger alphabet: of size m. The probability distribution is $(1, 2, .., m)/l$, where $l = \sum_{i=1}^{m} i = m(m + 1)/2$.

In Fig. 10 we see that it is a very rough bound. We would expect that ΔH grows approximately in linear way with respect to the size of alphabet, while in fact it seems to have a better behavior.

4.3 Combining with encryption

We will now briefly discuss using ANS to simultaneously encrypt the data or as a part of a cryptosystem. While standard cryptography usually operates on constant length bit blocks, encryption based on entropy coder has advantage of using blocks which lengths vary in a pseudorandom way. The inability to directly divide the bit sequence into blocks used in single steps makes cryptoanalysis much more difficult.

As it was mentioned in Section 3.5, the behavior of the ANS state is chaotic - if someone does not know the exact decoding tables, he would quickly loose the information about the current state. While decoding from some two neighboring states, they often correspond to a different symbol and so their further behavior will be very different (asymmetry). In comparison, in arithmetic coding nearby values are compressed further - remain nearby.

The crucial property making ANS perfect for cryptographic applications is that, in contrast to arithmetic coding, it has huge freedom of choice for the exact coding - every symbol distribution defines a different encoding. We can use a pseudorandom number generator (PRNG) initialized with a cryptographic key to choose the exact distribution, for example by disturbing the precise initialization algorithm. One way could be instead of choosing the pair with the smallest v by `getmin` operation, use the PRNG to choose between s which would be originally chosen and the second one in this order. This way we would get a bit worse precision and so ΔH, but we have $2^{l(b-1)}$ different possibilities among which we choose the encoding accordingly to the cryptographic key.

This philosophy of encryption uses the cryptographic key to generate a table which will be later used for coding. Observe that if we would make the generation of this table more computationally demanding, such approach would be more resistant to brute force attacks. Specifically, in standard cryptography the attacker can just start decryption with succeeding key to check if this key is the correct one. If we enforce some computationally demanding task to generate decoding tables (requiring for example 1ms of calculations), while decoding itself can be faster as we mainly use the table, checking large number of cryptographic keys becomes much more computationally demanding - this encryption philosophy can be made more resistant to unavoidable: brute force attacks.

The safeness of using ANS based cryptography depends on the way we would use it and the concrete scenario. Having access to both input and output of discussed automates, one usually could deduce the used encoding tables - there would be needed some additional protection to prevent that. However, if there was a safe PRNG used to generate them, obtaining the tables does not compromise the key. So initializing PRNG with the cryptographic key and additionally some number, would allow to choose many independent encodings for this key.

Beside combining entropy coding with encryption, ANS can be also seen as a cheap and simple building block for stronger cryptographic systems, like a replacement for the S-box. Example of such cryptography can be: use the PRNG to choose an intermediate

symbol probability distribution and two ANS coders for this distribution. Encoding of a bit sequence would be using the first encoder to produce a symbol in this distribution, and immediately use the second decoder to get a new bit sequence.

5 Conclusions

There was presented a way to improve memoryless prefix codes by adding some memory to create low state entropy coding automates. Their distance from Shannon entropy (ΔH) generally decreases like $1/l^2$ and increases linearly with the size of alphabet.

It is a general tool which main advantages while comparing with Huffman and arithmetic coding are:

- simple low state entropy coders, extremely close to the capacity - perfect for cheap and high throughput hardware implementations,
- switching encoder and decoder, we can encode a bit message into symbol sequence of chosen probability distribution, what might be required by some constrained channels,
- for larger number of states, such automate can process a symbol from large alphabet per cycle (table use) - for high throughput applications,
- the huge freedom of choosing the exact encoder and chaotic state behavior makes it perfect to simultaneously encrypt the data or as an inexpensive nonlinear building block of cryptosytems.

The disadvantages are:

- while directly using alphabet larger than 2, it requires initialization process for assumed probability distribution. The memory and computational cost is proportionally to the number of states. If distribution varies, this process has to be repeated,
- the decoding is in opposite direction to encoding, what needs storing the final state and can be an inconvenience, especially while adaptive applications.

There have remained many questions regarding this method, like how to optimally choose symbol distribution, finding better general formula for ΔH, analyzing situation after adding correlations to the source. Another research questions are related with its cryptographic capabilities - to understand how to use it, combine with other methods to obtain required level of cryptographic security.

Finally we should understand fundamental questions like if this method is optimal while using finite state automate for entropy coding, maybe find other low state entropy coders. Especially those able to encode and decode in the same direction - range/arithmetic coding is able to do it and can be seen as finite state automate, so the natural question is if it can be done better?

References

[1] T.M. Cover, *Enumerative source encoding*, IEEE Trans. Inf. Theory, vol. IT-19, pp. 73-77, Jan. 1973,

[2] J. Duda, *Optimal encoding on discrete lattice with translational invariant constrains using statistical algorithms*, arXiv:0710.3861,

[3] J. Duda, *Asymmetric numerical systems*, arXiv:0902.0271,

[4] J. Duda, *Data Compression Using Asymmetric Numeral Systems*, Wolfram Demonstration Project,
http://demonstrations.wolfram.com/author.html?author=Jarek+Duda ,

[5] D.A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the I.R.E., September 1952, pp 1098-1102,

[6] J.J. Rissanen, *Generalized Kraft inequality and arithmetic coding*, IBM J. Res. Develop., vol. 20, no. 3, pp. 198-203, May 1976,

[7] M. Mahoney, Data Compression Programs website, http://mattmahoney.net/dc/ ,

[8] G.N.N. Martin, *Range encoding: an algorithm for removing redundancy from a digitized message*, Video and Data Recording Conf., Southampton, England, July 1979,

[9] W. Szpankowski, *Asymptotic Average Redundancy of Huffman (and Other) Block Codes*, IEEE Trans. Information Theory, **46** (7) (2000) 2434-2443.