New Compression Schemes for Natural Number Sequences

Sapir Asraf[1], Shmuel T. Klein[2], and Dana Shapira[1]

[1] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
asrafsapir@gmail.com, shapird@g.ariel.ac.il
[2] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

Abstract. Elias and Fano independently proposed a quasi-succinct representation for monotonic integer sequences. In case the standard deviation is high, we suggest using the well known C_γ code instead of the Unary code used by their solution. In case the integers are similar, not necessarily forming a monotonic sequence, we propose to apply the Haar transform as a preprocessing stage, to achieve additional savings. Experimental results support the additional savings carried out by using our method.

Keywords: lossless compression, universal codes, the Haar transform

1 Introduction

Fixed length codes, such as the *American Standard Code for Information Interchange* ASCII code, are the most popular method to store data, as they provide simplicity, direct access and the possibility for fast retrieval. When compression performance is of interest, variable length codes are usually more effective. Obviously, the codes should be *Uniquely Decipherable* (UD), meaning that there is no ambiguous decoding. In case no codeword is a prefix of any of the other codewords, the code is often called a *Prefix-free Code*, and such a code is also UD. The restriction to prefix-free codes does not hurt the compression performance. Famous prefix-free variable length codes are, for instance, Huffman [12], Elias [5] and Fibonacci [7] codes.

Elias [5] proposed mainly two *fixed*, universal, prefix codeword sets, named C_γ and C_δ, in which any integer x is represented by a binary codeword composed of two parts. The first part listing the number of bits in the binary representation of x, and the second storing the standard binary representation itself without its leading 1-bit. While the first part is encoded by C_γ using the Unary encoding, C_δ uses C_γ. There is no difference between C_γ and C_δ in the second part. The expected codeword lengths are within twice the optimal average codeword length for the same underlying source for C_γ, and only a log log factor away from optimal for C_δ. More precisely, C_γ requires $2\lfloor \log x \rfloor + 1$ and C_δ necessitates $\lfloor \log x \rfloor + 1 + 2\lfloor \log(\lfloor \log x \rfloor + 1) \rfloor$ bits to encode the number x.

A classical way to encode a monotonic set of integers is *differential encoding*, also called *gap encoding*. In this method, instead of encoding the original set $0 \leq x_1 \leq x_2 \leq \cdots \leq x_n \leq U$, the set of differences is encoded. If the sequence of differences is encoded by C_δ, then the number of bits used is no more than $n \log \frac{U}{n} + 2n \log\log \frac{U}{n} + O(n)$, which is close to the zero-order entropy of a bit-vector of size U with n 1-bits [14].

There are quite a few motivations for the interest in monotonic sequence, an *Inverted Index* is one of them. This is a powerful data structure commonly used in Information Retrieval to enhance the processing time of search engines. Given

Sapir Asraf, Shmuel T. Klein, Dana Shapira: New Compression Schemes for Natural Number Sequences, pp. 1–10.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9 © Czech Technical University in Prague, Czech Republic

a collection of documents, the inverted index is the list of documents where each element of the collection occurs in, possibly including the frequency as well as the exact positions of the element within each document. For a text T, the inverted index stores for each element w with n_w occurrences, the positions $x_1 < x_2 < \cdots < x_{n_w}$ within T where w occurs. As this list is usually given in order, the resulting sequence of integers is increasing.

Increasing sequences can also be found in *Compressed Suffix Arrays.* A suffix array (SA) for $T\$$, where T is a string of length n over Σ and $\$ \notin \Sigma$, is an array $SA[0:n-1]$ of the indices of the suffixes of $T\$$, stored in lexicographical order. Grossi and Vitter [10] improve the space requirements of a suffix array by decomposing it based on the *neighbor function* Φ. It has been shown that the values of Φ at consecutive positions referring to suffixes that start with the same symbol must be increasing. The implementation of CSA used in [1,13] applies differential encoding on the neighbor function. Improved compression results were proposed by Gog et al. [8] who suggest using the *Elias-Fano* encoding for storing the increasing Φ values of the CSA.

Our paper is constructed as follows. Section 2 recalls the details of Elias-Fano codes, and suggests a variant that uses $C\gamma$ rather than the Unary code used by Elias-Fano. We show that the original Elias-Fano is suitable for homogeneous series, while the new variant is effective for series with higher standard deviation. Section 3 then suggests the Haar transform as a preprocessing stage in order to convert homogeneous numbers to a series which is suitable for the new C_γ variant. Experimental results presented in Section 4 then support the savings of the proposed method.

2 Quasi-succinct representation for monotone sequences

Gap encoding can be used in order to compress inverted indices. Instead of encoding the non-decreasing list of integers $0 \le x_1 \le x_2 \le \cdots \le x_n$, directly pointing to the ordered set of documents, the differences $d_1 = x_1,\ d_2 = x_2 - x_1,\ \ldots,\ d_n = x_n - x_{n-1}$ are encoded, usually by universal codes such as Elias, Golomb [9] or Rice codes.

Elias [4] and Fano [6] independently proposed an efficient encoding method for representing a non-decreasing sequence $\mathcal{X}$ of positive integers

$$\mathcal{X} = \{0 \le x_1 \le x_2 \le \cdots \le x_n \le U\},$$

where U is a given upper bound on x_n, possibly equal to x_n.

The sequence $\mathcal{X}$ is represented by two separate bit-vectors $\mathcal{L}(\ell)$ and $\mathcal{U}(\ell)$, for storing the ℓ lower bits, and the differences between successive values of the remaining upper bits of each x_i, respectively. More precisely, the ℓ least significant bits of each $x_i \in \mathcal{X}$, which are $x_i \bmod 2^\ell$, are stored sequentially in $\mathcal{L}(\ell)$ in the same order as they appear in $\mathcal{X}$. The value of the binary representation of the remaining bits of each $x_i \in \mathcal{X}$, that is, the values $y_i = \lfloor \frac{x_i}{2^\ell} \rfloor$ are then considered, and the differences between adjacent values $\Delta(i) = y_i - y_{i-1}$, $1 \le i \le n$, are computed, setting $y_0 = 0$. $\mathcal{U}(\ell)$-UNARY stores these differences $\Delta(i)$ in the same order as in $\mathcal{X}$ in a Unary encoding, that is, representing the integers $1, 2, 3, \ldots, i$ respectively by $1, 01, 001, \ldots, 0^{i-1}1$. Elias-Fano's method defines ℓ to be equal to $\max\{0, \lfloor \log(\frac{U}{n}) \rfloor\}$. A similar encoding of the indices of 1-bits in a sparse bit-vector, in which the sequence $\mathcal{U}(\ell)$ is replaced by a bit-vector, is described in [2].

Table 1 displays the representation of the Elias-Fano code applied on the monotonic sequence example 2, 3, 10, 16, and 52. The first and second lines of Table 1

give the sequence $\mathcal{X}$ and their binary representation $\mathcal{B}(x_i)$. According to Elias-Fano, $\ell = \max\{0, \lfloor \log(\frac{U}{n}) \rfloor\} = 3$ for our example, and the third line presents the lower bits vector $\mathcal{L}$ for $\ell = 3$. The next block of four lines are the stages for constructing $\mathcal{U}$-UNARY for this example, given at the end of this block. The following two lines, headed by $\mathcal{B}\big(\lfloor \frac{x_i}{2^\ell} \rfloor\big)$ and $\lfloor \frac{x_i}{2^\ell} \rfloor$, are the remaining bits in each x_i and their corresponding values after their 3 lower bits have been removed. The line headed by $\Delta(i)$ is the differences between adjacent values of the previous line. Elias-Fano uses 26 bits in total. The last five lines of Table 1 are explained below.

	x_1	x_2	x_3	x_4	x_5
$\mathcal{X}$	2	3	10	16	52
$\mathcal{B}(x_i)$	10	11	1010	10000	110100
$\mathcal{L}(3)$	010	011	010	000	100
$\mathcal{B}\big(\lfloor \frac{x_i}{2^\ell} \rfloor\big)$	0	0	1	10	110
$\lfloor \frac{x_i}{2^\ell} \rfloor$	0	0	1	2	6
$\Delta(i)$	0	0	1	1	4
$\mathcal{U}(3)$-UNARY	1	1	01	01	00001
$\mathcal{U}(3)$-C_γ	1	1	010	010	00101
$\mathcal{L}(2)$	10	11	10	00	00
$\mathcal{B}\big(\lfloor \frac{x_i}{2^\ell} \rfloor\big)$	0	0	10	100	1101
$\lfloor \frac{x_i}{2^\ell} \rfloor$	0	0	2	4	13
$\Delta(i)$	0	0	2	2	9
$\mathcal{U}(2)$-C_γ	1	1	011	011	0001010

Table 1. Quasi Succinct Encoding [4] for the sequence 2, 3, 10, 16 and 52

Exactly $\ell \cdot n$ bits are used for storing the lower bits vector $\mathcal{L}(\ell)$. Next, we compute the number of bits used by the upper bits vector $\mathcal{U}(\ell)$-UNARY. The Unary code records the values $y_i - y_{i-1} = \frac{x_i}{2^\ell} - \frac{x_{i-1}}{2^\ell}$. If this difference is c, then x_i is larger than x_{i-1} by at least $c \cdot 2^\ell$. The total differences can obviously not be larger than $\frac{x_n}{2^\ell}$, the latter being bounded in case the definition for ℓ is used, explained as follows.

$$\left\lfloor \frac{x_n}{2^\ell} \right\rfloor \leq \left\lfloor \frac{U}{2^\ell} \right\rfloor \leq \frac{U}{2^\ell} = \frac{U}{2^{\max\{0, \lfloor \log(U/n) \rfloor\}}}$$

If there exists an integer k so that $\frac{U}{n} = 2^k$ then $\frac{U}{2^{\max\{0, \lfloor \log(U/n) \rfloor\}}} = n$. Otherwise, $\lfloor \log(U/n) \rfloor = \lceil \log(U/n) \rceil - 1$, and $\frac{U}{2^{\max\{0, \lfloor \log(U/n) \rfloor\}}} \leq 2n$.

Each Unary codeword requires a single 1-bit, and each 0-bit within the Unary codeword represents an increase by 2^ℓ. At most n 1s and $2n$ 0s are written in the Unary representation, that is, 3 bits per integer x_i. This concludes that the representation of Elias-Fano uses at most $2 + \lceil \log(\frac{U}{n}) \rceil$ bits per element. Elias [4] proves that the Elias-Fano representation is close to optimal as the information theoretical lower bound for

a monotonic sequence of n integers is

$$\left\lceil \log \binom{U+n}{n} \right\rceil \approx n \log \left(\frac{U+n}{n} \right).$$

Although, Elias-Fano's encoding is considered *quasi-succinct*, that is, close to the optimal representation, which is the information theoretical bound [16], there is still place for improvements by replacing the Unary code by C_γ, as the former code is costly for large integers. The Unary encoding uses $i+1$ bits to encode the integer i, $i \geq 0$, i.e., the codeword for the integer i is $\mathtt{0}^i\mathtt{1}$. The ith codeword for C_γ refers to the binary representation of $i+1$, denoted by $\mathcal{B}(i+1)$, as the value zero may also be encoded. The number of bits in $\mathcal{B}(i+1)$ is encoded using its Unary form, followed by $\mathcal{B}(i+1)$ after the preceding 1-bit has been removed. The first several codewords of Unary and C_γ are

	0	1	2	3	4	5	6	7	8
U	1	01	001	0001	00001	000001	0000001	00000001	000000001
C_γ	1	01 0	01 1	001 00	001 01	001 10	001 11	0001 000	0001 001

where blanks are inserted between the unary and the binary parts for readability. Only for the codewords corresponding to values 1 and 3 are Elias' C_γ codewords longer than those of the Unary code; for all other values, C_γ is preferable to the Unary code. We therefore propose a different variant of the Elias-Fano encoding, which is especially useful for non-uniform monotonic sequences having large standard deviation.

The sequence $\mathcal{X}$ is still represented by two bit vectors $\mathcal{L}(i)$ and $\mathcal{U}(i)$ for storing, respectively, the i lower bits and the differences between successive values of the remaining upper bits of each x_i, but this time we shall not fix the number of bits i in advance and rather let it vary from 0 to $\ell = \max\{0, \lfloor \log(\frac{U}{n}) \rfloor\}$. Using the example of Table 1, the line headed $\mathcal{U}(3)$-C_γ refers to the case of $i = 3$ and presents the corresponding $\mathcal{U}(3)$ for C_γ. The lower bits vector $\mathcal{L}(3)$ remains the same, for a total of 28 bits, instead of 26 used by the original Elias-Fano. However, the representation for $i = 2$ uses only 25 bits, shown by the bottom block of Table 1. The lower bits vector $\mathcal{L}(2)$ uses 10 bits, and $\mathcal{U}(2)$-C_γ uses additional 15 bits, less than the 26 bits used by Elias-Fano.

ℓ	0	1	2	3	4	5	6
Elias-Fano-C_γ	35	34	31	36	37	38	41

Table 2. Elias-Fano-C_γ for the sequence $2, 3, 10, 16, 520$ where the original Elias-Fano uses 43 bits

The introduction of the Elias-Fano-C_γ variant was, however, not suggested for the savings of merely a single bit, and is rather suitable for sequences with larger standard deviation. Consider the same example in which the last element has been changed to 520. The standard deviation grows from 18.41 for the first one, to 204.96 for this new example. Elias-Fano-Unary requires 43 bits for the updated sequence, while Elias-Fano-C_γ only needs 31 bits, which is attained for $\ell = 2$. Table 2 gives the total number of bits required for encoding the sequence 2, 3, 10, 16, 520 by the new version, as a function of ℓ. It is interesting to see that the storage for all values of ℓ needs less space than the original Elias-Fano coding.

We thus see that there is an advantage for using C_γ for sequences with high variability. Obviously, for general data, the logarithmic encoding of C_γ will be preferable to the linear encoding of Unary, but for very uniform data, the differences encoded by $\mathcal{U}$ in the Elias-Fano scheme will tend to consist mainly of very small integers, for which the Unary variant is not so bad. In order to improve also the compression of more homogeneous integer sets, we apply an idea used repeatedly in other data compression applications, namely that of using a reversible transformation of the original input to produce an equivalent sequence that is more compressible. This has been used by applying the Burrows-Wheeler transform (BWT) for the compression of textual and other data [3], or the discrete cosine transform in lossy image compression by JPEG [15]. In our case, we aim at causing a set of integers to be less homogeneous. It turns out that the existence of an extreme element in a sequence is typical for the output of the Haar transform [11], suggested as a preprocessing stage in the next section.

3 The Haar Transform

The Haar wavelet transform, is a simple discrete transform, used in practical encoding applications such as the compression of digitized sound and images. Here it is applied for lossless compression of integer sequences. The Haar transform uses the basic scale function $\phi(t)$, and the basic wavelet function $\psi(t)$ defined as follows.

$$\phi(t) = \begin{cases} 1, & 0 \le t < 1 \\ 0, & otherwise. \end{cases} \qquad \psi(t) = \begin{cases} 1, & 0 \le t < 0.5 \\ -1, & 0.5 \le t < 1. \end{cases}$$

A target function $f(t)$ is approximated by an infinite linear combination of $\phi(t-k)$ and $\psi(2^j t - k)$, where the parameter k assumes all possible, positive, negative and zero, integer values:

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \phi(t-k) + \sum_{k=-\infty}^{\infty} \sum_{j=0}^{\infty} d_{j,k} \psi(2^j t - k),$$

where c_k and $d_{j,k}$ are constants. The function is transformed to a low resolution average $\phi(t)$ and the high resolution detail $\psi(t)$. In this research we are interested in a particular non-normalized Haar transform, and refer to its matrix representation.

The Haar transform is related to a matrix of order $2^k \times 2^k$ for $k \ge 1$. The non-normalized Haar matrix $\mathcal{H}_2$ of order 2×2 is $\mathcal{H}_2 = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. The Haar matrix $\mathcal{H}_{2^k}$ of order $2^k \times 2^k$ is defined recursively by $\mathcal{H}_{2^k} = \begin{pmatrix} \mathcal{H}_{2^{k-1}} \otimes (1,1) \\ I_{2^{k-1}} \otimes (1,-1) \end{pmatrix}$, where $\otimes$ is the *Kronecker product* defined for an $n \times m$ matrix A and a $t \times r$ matrix B as the $nt \times mr$ matrix $A \otimes B$ obtained by:

$$\text{if} \quad A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix} \quad \text{then} \quad A \otimes B = \begin{pmatrix} a_{1,1}B & \cdots & a_{1,n}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \cdots & a_{m,n}B \end{pmatrix}.$$

Given a sequence of 2^k values $a_1, a_2, \ldots, a_{2^k}$, the Haar transform computes, for each pair of values a_{2i-1} and a_{2i}, $i = 1, \ldots, 2^{k-1}$, the quantities

$$avg(i) = \frac{a_{2i-1} + a_{2i}}{2} \qquad \text{and} \qquad \Delta(i) = \frac{a_{2i-1} - a_{2i}}{2}.$$

The resulting sequence is composed of the averages $avg(1), avg(2), \ldots, avg(2^{k-1})$, followed by the half-differences, $\Delta(1), \Delta(2), \ldots, \Delta(2^{k-1})$, and it is of the same length as the input sequence. The 2^{k-1} averages are recursively transformed into 2^{k-2} new averages followed by 2^{k-2} half-differences, and so on until only a single element remains. The produced single value followed by the 2^{j-1} half-differences obtained from all stages, $j = 2, \ldots k$, are concatenated to form the final transformed elements. Note that this single value, together with the sequences of half-differences, is sufficient to reconstruct the original sequence, so the Haar transform is reversible.

As an example consider the sequence $\mathcal{X} = \{1840, 1680, 1632, 1504, 1536, 1472, 1360, 1328\}$. The Haar transform applied on $\mathcal{X}$ is presented in Figure 1 resulting in the non-increasing sequence $\mathcal{H}(\mathcal{X}) = \{1544, 120, 96, 80, 80, 64, 32, 16\}$. The original series is given on the first line. The partition into pairs is depicted by curly braces, and their average is presented on the following line. The elements contributed to the resulting Haar vector, are shown in gray. The last line is the Haar transform outcome.

Algorithm 1: *Haar-C_γ*

HAAR-$C_\gamma(x_1, \ldots, x_{2^k})$

1 $(h_1, \ldots, h_{2^k}) \leftarrow$HAAR$(x_1, \ldots, x_{2^k})$
2 $U \leftarrow h_1$
3 $\ell \leftarrow \max\{0, \lfloor \log(\frac{U}{2^k}) \rfloor\}$
4 Encode $(h_{2^k}, \ldots, h_1)$ using ELIAS-FANO-$C_\gamma(i)$ for $0 \le i \le \ell$ choosing i that results with minimum number of bits

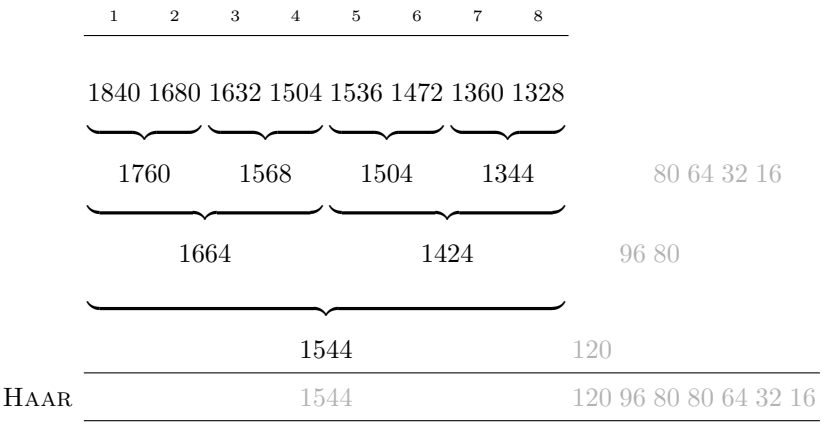

Figure 1. The Haar Transform for $\mathcal{X} = \{1840, 1680, 1632, 1504, 1536, 1472, 1360, 1328\}$

The resulting output of the Haar transform of Figure 1 is quite typical: a dominant first coefficient followed by others that are smaller by orders of magnitude, and most

importantly, with higher standard deviation than the original series. When the Haar transform is applied to an image, the averages of the disjoint successive pairs are commonly named the *coarse resolution* of the input image, while the differences of the pairs are called the *detail coefficients*. The Haar transform is effective for correlated pixels, as the coarse representation will resemble the original pixels, while the detail coefficients will be small. The small values tend to be more compressible than the original ones, and several compression techniques can be applied such as *Run-Length Encoding*, *Move-To-Front* and Huffman encoding for lossless compression, possibly adding quantization for lossy compression. For more details on the Haar transform we refer the reader to the book of Salomon [15].

In this research we suggest to apply Algorithm 1 in case the input integer series consists of similar numbers. Algorithm 1, which assumes that the input size is a power of 2, 2^k, starts by applying the Haar transform on the input sequence $(x_1, \ldots, x_{2^k})$ on line 1 and obtains the output sequence $(h_1, \ldots, h_{2^k})$ of the same length as a result. It then computes ℓ on line 3 as defined by Elias-Fano, and encodes the reverse sequence $(h_{2^k}, \ldots, h_1)$, to get an increasing sequence, with Elias-Fano-$C_\gamma(i)$, for i ranging from 0 to ℓ, choosing a value of i that results in the minimum number of bits.

Continuing our running example of Figure 1, we applied Algorithm 1 on the given sequence. The encoding of Haar-C_γ results in 62 bits, which was attained for $\ell = 3$. For comparison, Elias-Fano-Unary and Elias-Fano-C_γ on the *sorted* sequence of the original series gave 78 bits for $\ell = 7$ and 76 bits for $\ell = 6$, respectively. We also applied Elias-Fano-Unary on the resulting Haar vector, that, as noted above, is sorted for this example, which attained 76 bits for $\ell = 7$.

3.1 Encoding the Haar output using two blocks

In order to apply Algorithm 1, the Haar transform must result in a monotonic decreasing sequence, which is not necessarily the case. Algorithm 2 suggests the encoding by Elias-Fano-C_γ with only two different values for $\mathcal{U}(i)$-C_γ. That is, the sequence is partitioned into two buckets: the first containing only the first element, and all the others belonging to the second one. The corresponding values of $\mathcal{U}(i)$-C_γ are therefore all 0, so they can be omitted.

Interestingly, this encoding does not require a monotonic series as all coordinates, except the first, are written explicitly in the lower bits $\mathcal{L}$ array.

Algorithm 2: Bi-Haar-C_γ

Bi-Haar-$C_\gamma(x_1, \ldots, x_{2^k})$

1 $(h_1, h_2, \ldots, h_{2^k}) \leftarrow$Haar$(x_1, \ldots, x_{2^k})$
2 $m \leftarrow h_2$
3 $\ell \leftarrow \lfloor \log m \rfloor + 1$
4 Encode $(h_{2^k}, \ldots, h_1)$ using Elias-Fano-$C_\gamma(\ell)$, without encoding zeros in $\mathcal{U}$

Applying Algorithm 2 on our running example, $m = 120$, and the coefficients are encoded by $\lfloor \log m \rfloor + 1 = 7$ bits, for a total of 56 for $\mathcal{L}(7)$. The upper bits vector $\mathcal{U}(7)$-C_γ needs only to express the coarse coefficient, as the seven detail coefficients are with upper bit 0, which can be omitted for this method. In our example, the coefficient $\lfloor \frac{1544}{2^7} \rfloor = 12$, which is encoded by 13 bits in a Unary code, and by only 7 bits in C_γ, for a total of 69 and 63 bits, respectively.

3.2 Ensuring that the Haar transform results in integers

The Haar transform repeatedly computes the averages of number pairs in the input series, which may, obviously, produce non-integer numbers. Since Elias-Fano methods are restricted to integers, Algorithm 3 presents a variant of the Haar transform that makes sure the outcome consists only of integers. This is done by prepending a bit b_i, $1 \leq i \leq 2^k$ to each of the differences, indicating whether the corresponding average is exact or has been rounded. In fact, b_i is a parity bit as used in error correcting codes: in case the corresponding sum is even, b_i is set to 0, and if it is odd, $b_i = 1$. Concatenation is denoted by $\cdot$. The additional bits enables the reversibility of the Haar transform. The algorithm gets as input a sequence of $n = 2^k$ integers for some $k \geq 1$, and returns another sequence of n integers, the first being the (rounded) overall average, followed by $n - 1$ differences.

Algorithm 3: *Integer-Haar-C_γ*

INTEGER-HAAR-$C_\gamma(x_1, \ldots, x_{2^k})$
for $i \leftarrow 1$ **to** 2^{k-1} **do**
  $b_{2^{k-1}+i} \leftarrow (x_{2i} - x_{2i-1}) \bmod 2$
  $h_{2^{k-1}+i} \leftarrow \lfloor \frac{1}{2}(x_{2i} - x_{2i-1}) \rfloor$
  $z_i \leftarrow \lfloor \frac{1}{2}(x_{2i} + x_{2i-1}) \rfloor$
if $k = 1$ **then**
  return $(z_1, b_2 \cdot h_2)$
else
  $(y_1, \ldots, y_{2^{k-1}}) \leftarrow$HAAR$(z_1, \ldots, z_{2^{k-1}})$
  return $(y_1, \ldots, y_{2^{k-1}},\ b_{2^{k-1}+1} \cdot h_{2^{k-1}+1}, \ldots, b_{2^k} \cdot h_{2^k})$

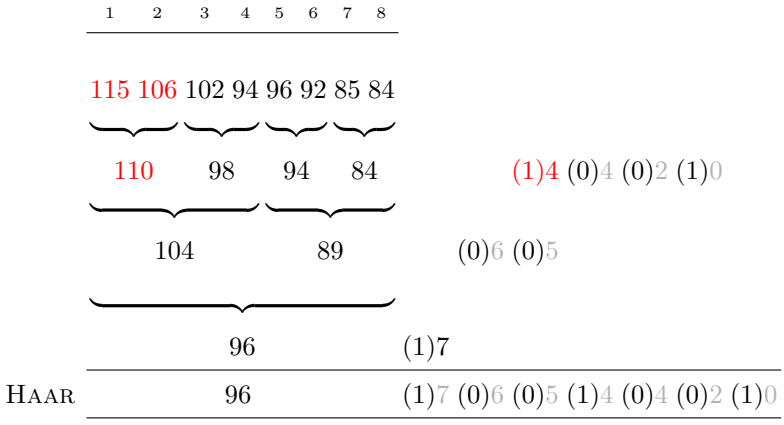

Figure 2. Haar Transform with two buckets

As example, consider the following sequence of integers $\mathcal{X} = 115, 106, 102, 94, 96, 92, 85, 84$, depicted in Figure 2. Each difference d on the right hand side is now preceded by a parity bit b in parentheses, which is set to 1 if and only if the corresponding average value a on the left hand side has been rounded, that is, the

sum of the two integers a' and a'' of the previous iteration, was odd, see the example in red in Figure 2. The reversibility means that we can recover a' and a'' from a, d and b. Indeed:

$$a' = a + d + b \qquad \text{and} \qquad a'' = a - d.$$

4 Experimental Results

In order to evaluate our proposed method, we considered randomly generated sequences of 256 elements. We defined U to be the largest element in the sequence and generated the Elias-Fano-C_γ encoding for varying values of ℓ from 1 to 30. All sequences presented a similar behavior, Figure 3 and Figure 4 depict the compression results of a typical representative.

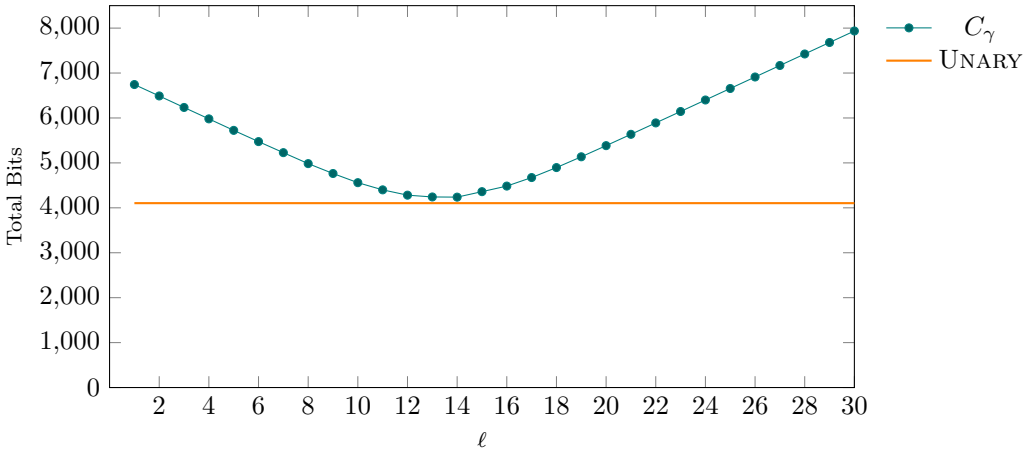

Figure 3. Elias-Fano-Unary vs. Elias-Fano-C_γ encoding for uniform random generated monotonic sequence of 256 elements

The general case is shown in Figure 3. Elias-Fano-Unary gives the best result, 4104 bits. The best encoding for Elias-Fano-C_γ is only slightly larger, 4238 bits, for $\ell = 14$, and it deteriorates for other values of ℓ. To get examples of more biased input sequences, the test was repeated again with randomly generated sequences, but to each of which one extreme element has been adjoined, thereby simulating the series handled by JPEG or after having applied the Haar transform. The corresponding graph of a typical example appears in Figure 4. Elias-Fano-Unary stores the input sequence using 6,144 bits with $\ell = 22$, while many Elias-Fano-C_γ values were lower, with a minimum achieved of 1,698 bits, for $\ell = 3$.

References

1. E. Benza, S. T. Klein, and D. Shapira: *Smaller compressed suffix arrays.* The Computer Journal, 2020.
2. A. Bookstein and S. T. Klein: *Compression of correlated bit-vectors.* Inf. Syst., 16(4) 1991, pp. 387–400.

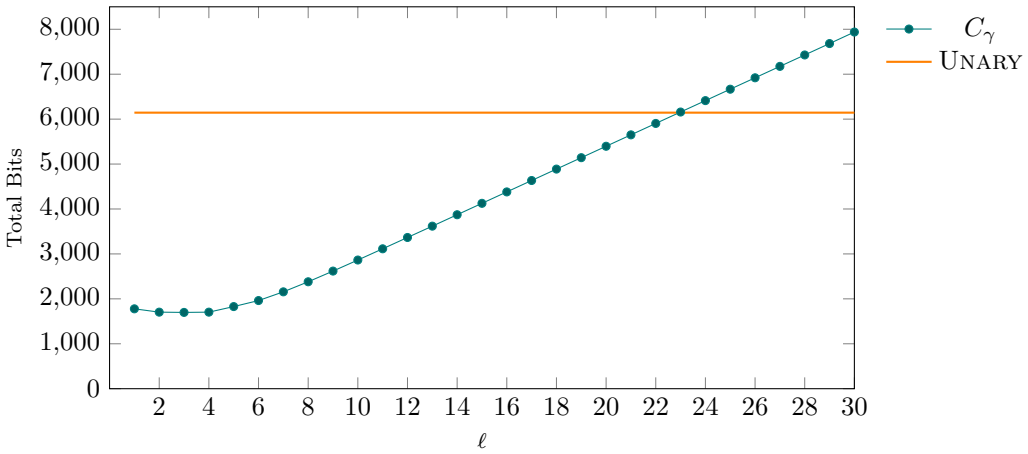

Figure 4. Elias-Fano-Unary vs. Elias-Fano-C_γ encoding for a random generated monotonic sequence of 256 elements, with an extreme element

3. M. Burrows and D. J. Wheeler: *A block sorting lossless data compression algorithm*, in SRC Technical Report **124**, Digital Equipment Corporation, Palo Alto, CA, 1994.
4. P. Elias: *Efficient storage and retrieval by content and address of static files.* J. ACM, 21(2) 1974, pp. 246–260.
5. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Trans. Information Theory, 21(2) 1975, pp. 194–203.
6. R. Fano: *On the Number of Bits Required to Implement an Associative Memory*, Computation Structures Group Memo, MIT Project MAC Computer Structures Group, 1971.
7. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.
8. S. Gog, A. Moffat, and M. Petri: *CSA++: fast pattern search for large alphabets*, in Proc. 19th Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, January 17-18, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2017, pp. 73–82.
9. S. W. Golomb: *Run-length encodings (corresp.).* IEEE Trans. Inf. Theory, 12(3) 1966, pp. 399–401.
10. R. Grossi and J. S. Vitter: *Compressed suffix arrays and suffix trees with applications to text indexing and string matching.* SIAM Journal on Computing, 35(2) 2005, pp. 378–407.
11. A. Haar: *Zur Theorie der orthogonalen Funktionensysteme.* Mathematische Annalen, 69(3) 1910, pp. 331–371.
12. D. A. Huffman: *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 40(9) 1952, pp. 1098–1101.
13. H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich: *A practical implementation of compressed suffix arrays with applications to self-indexing*, in Proceeding of the Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26–28 March, IEEE Computer Society, Los Alamitos, CA, 2014, pp. 292–301.
14. G. Navarro: *Compact Data Structures - A Practical Approach*, Cambridge University Press, Cambridge UK, 2016.
15. D. Salomon, G. Motta, and D. Bryant: *Data Compression: The Complete Reference*, Molecular biology intelligence unit, Springer London, 2007.
16. S. Vigna: *Quasi-succinct indices*, in Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013, S. Leonardi, A. Panconesi, P. Ferragina, and A. Gionis, eds., ACM, 2013, pp. 83–92.