Advance Access publication on June 5, 2008 doi:10.1093/comjnl/bxn030

Tournament Coding of Integer Sequences

JUKKA TEUHOLA*

Department of Information Technology, University of Turku, Joukahaisenkatu 3-5, FI-20014 Turku, Finland
**Corresponding author: teuhola@it.utu.fi*

A new, simple non-statistical source coding technique for sequences of integers is suggested. The method is based on a tournament scheme, with the sequence arranged into pairs, where maxima ('winners') are encoded recursively, and minima are encoded by semi-fixed-length codes using the related maxima to bound the code lengths. In the experiments, tournament coding has outperformed the other non-statistical methods (gamma, delta, Fibonacci and interpolative coding) for uniform distribution of numbers. Also for non-uniform distributions the method is quite competitive.

Keywords: source coding; interpolative coding; data compression

Received 5 October 2007; revised 28 February 2008

1. INTRODUCTION

Compression is an essential way of saving resources in storage and transmission of many sorts of data. The highest savings in practice are obtained by *lossy* compression of image, audio and video data, but also *lossless* text compression has a long tradition, with a large set of developed techniques, see for example [1, 2]. Lossless compression guarantees exact recovery of the original data, while with lossy methods the decompression result is an approximation of the original.

The compression process is usually divided into two main phases: *modelling* and *coding*. Modelling typically produces a set of (possibly transformed) items, together with their probability distribution, as the input for coding. In many modelling methods, the source data is transformed into a sequence of small integers that can be encoded more compactly than the original items. Examples of such modelling techniques include the following:

- In *Run-length coding* of (clustered) binary strings, such as black-and-white images, the lengths of alternating runs of zeroes and ones (black and white pixels) constitute the numbers to be encoded. The well-known *Golomb code* [3] was developed for run-length coding. A practical example is facsimile compression.
- *Lossless image compression* is often based on the *prediction* of pixel values (colours), and the deviations from predictions are the numbers to be encoded. In this case also negative numbers can appear.
- *Lossy image compression* methods usually apply a transformation to the image data. For example, in the JPEG [4] standard, a discrete cosine transform is applied to blocks of pixels, producing blocks of quantized coefficients — again numbers to be encoded.
- In an *inverted index*, an inverted list consists of an ascending sequence of location pointers, related to a keyword. The differences of subsequent pointers constitute the sequence of integers to be encoded. This application has been the motivation behind *interpolative coding* [5, 6].
- *Burrows–Wheeler transform* [7] is a clever way of transforming text (or any string) by block-sorting, producing a clustered permutation of the original symbols. Another transform, typically *move-to-front* (MTF) technique, produces the final numbers (=MTF indices). The method has been developed further by many researchers. Alternative ways of coding the MTF indices were studied, e.g. in [8], see also [9]. The popular *bzip2* compression utility is based on the Burrows–Wheeler transform.

The realization of the second step of compression, namely the actual encoding, depends on a few preferences. If compression efficiency is emphasized, then *statistical coding* (also called *entropy coding*) techniques are usually recommended. The main options are *Huffman coding* [10] and *arithmetic coding* [11, 12]. The former is optimal [9] if distinct codewords are required for integers. The latter reaches the

information-theoretic lower bound, *entropy*, by using a fractional number of bits per integer.

Traditional Huffman coding has the drawback that a codebook must be generated in a preliminary phase, which first determines the integer frequencies. What is worse, for a large range of integers to be encoded, the codebook can become quite big; in the extreme case every integer occurs only once and occupies yet one entry in the codebook, which must be stored/transmitted to the decoder. *Adaptive Huffman coding* avoids this, but needs a dynamic probabilistic model, the maintenance of which during coding is cumbersome. For a large range of integers, dynamically gathered statistics can be quite unreliable. In *adaptive arithmetic coding*, the maintenance of the model is more flexible, but the method is generally rather slow.

Thus, there is clearly room for fast, non-statistical techniques for compact coding of arbitrarily large integers. Again, there are options: We can encode the integers independently by using always the same codeword for a certain integer. This is also called *static coding* of integers; examples are *non-parametric gamma* and *delta codes* [13], *Fibonacci code* (also called Zeckendorf code) [14, 15], and *parametric Golomb* [3] and *exp-Golomb* [16] codes. For a survey of universal codes, see [17]. Special speed-optimized techniques for coding are, for example, *byte-oriented coding* [18] and *word-aligned* coding [19]. They suffer from a small penalty in compression gain, but support partial decoding. Another main option is to choose the codewords of integers relative to other near-by integers in the sequence. *Interpolative coding* [5, 6] does just this, and the tournament coding, suggested here, as well.

To summarize, our task is to develop a compact representation for a sequence of integers from an arbitrary range $[0, u]$. Notice that the restriction to non-negative integers is not severe. A simple mapping $x' = 2x$ if x is non-negative, and $x' = -2x - 1$ if x is negative, is reversible and produces a non-negative range. In fact, any finite subset of an enumerable set, such as rational numbers, can be handled.

The paper continues by explaining the coding of bounded integers in Section 2. Section 3 reviews interpolative coding, which was the starting point of the current development and constitutes also its main 'competitor'. Section 4 describes the main idea of tournament coding, while refinements are presented in Section 5. The compression gain is analysed in Section 6 and experimented in Section 7. The paper ends with some conclusions in Section 8.

2. CODING OF BOUNDED INTEGERS

Assume that we should encode an integer i from the interval $[a, b]$. If we have no other prior knowledge about the distribution of i, a natural solution is to apply *fixed-length coding*, i.e. each integer in $\{a, \ldots, b\}$ is assigned a codeword of $\lceil \log_2 m \rceil$ bits, where m denotes the number of integers in the interval, i.e. $m = b - a + 1$. Clearly, this is not optimal unless m is a power of 2. A better choice is to assign codewords of $\lfloor \log_2 m \rfloor$ bits to $m_1 = 2^{\lceil \log_2 m \rceil} - m$ integers, and codewords of $\lceil \log_2 m \rceil$ bits to the rest $m_2 = m - m_1$. This technique, which we here call *semi-fixed-length coding*, is well known, and mentioned, for example in [2, 5, 6]. There are an exponential number of possible assignments of semi-fixed-length codewords to the n integers, but four of them are reasonable and easily computable:

- *low-short*: Shorter codewords are assigned to the low-end integers.
- *high-short*: Shorter codewords are assigned to the high-end integers.
- *mid-short*: Shorter codewords are assigned to the mid-range integers.
- *mid-long*: Longer codewords are assigned to the mid-range integers.

TABLE 1. Four semi-fixed-length codes for numbers $0, \ldots, 5$.

Number	Low-short	High-short	Mid-short	Mid-long
0	10	000	000	10
1	11	001	001	000
2	000	010	10	001
3	001	011	11	010
4	010	10	010	011
5	011	11	011	11

Moffat and Stuiver call the mid-short alternative as *centred minimal binary code* [6]. Table 1 shows sample codeword assignments in the case of interval $0, \ldots, 5$. Note that the mid-short code table is always symmetric with respect to codeword lengths, but mid-long is not for odd m. Also note that in Table 1, only the codeword lengths are essential. Instead, the exact assignment of bits is immaterial, as long as the codewords are distinct, and no codeword is a prefix of another. Codewords should be selected in a way that enables simple programming in the case when the code table cannot be precomputed.

It is easy to avoid bit-by-bit processing in both encoding and decoding of any of the codes in Table 1, but the details are skipped here.

Even though statistical coding is not pursued, we may have some guess about non-uniformity of the item distribution. Thus we can try to improve the coding efficiency by selecting the code table, which assigns the shorter codewords to the more frequent items. This has been exploited for example in interpolative coding [6], where both mid-short and mid-long code tables are used, in different steps of the method. Also in tournament coding, choosing the best of the four alternatives will be considered.

3. REVIEW OF INTERPOLATIVE CODING

Interpolative coding [5, 6] is a simple, yet very efficient way of encoding sequences of integers. As for compression gain, it has been shown practically to be superior to all other non-parametric, non-statistical coding methods. Actually, the method was developed for inverted indexes where inverted lists consist of strictly ascending sequences of integers. Any (non-ascending) sequence can be converted to ascending by computing the cumulative values. The strictness of ascent is not relevant. The functioning of the method is now explained using an example. Assume that the integers to be encoded are:

$$\langle 4, 2, 0, 3, 5, 1, 2, 3 \rangle$$

These are converted to cumulative numbers:

$$\langle 4, 6, 6, 9, 14, 15, 17, 20 \rangle$$

The method first encodes the last element by using some universal coding scheme, such as gamma or delta code [13]. Now we (and the decoder) know that all values in the list are between 0 and 20. Especially, we choose the middle element 9 of the remaining sequence and encode it with $\lfloor \log_2(20 - 0 + 1) \rfloor = 4$ bits by using the mid-short coding scheme. (A reasonable guess is that the middle value is about half of the upper bound, so mid-short seems a good choice.) The process continues recursively: We know that all values in the first half of the sequence are between 0 and 9, so the middle element 6 can be encoded with $\lfloor \log_2(9 - 0 + 1) \rfloor = 3$ bits. The values in the upper half are between 9 and 20, and the middle element 15 is encoded with $\lfloor \log_2(20 - 9 + 1) \rfloor = 3$ bits. We proceed by repeated halving, either until reaching a subsequence of size one, or to the situation where the smallest and largest integer within the subsequence are the same. In the latter case, the original list contains all zeroes in this subsequence, and need not be encoded at all. This advantageous alternative is actually quite common in many compression applications.

As an introduction to tournament coding, we study another interpretation of interpolative coding. We build a binary tree of pairwise sums, with original integers as leaves. Figure 1 shows the tree of the above sequence. Only the encircled nodes (root and left siblings) need to be encoded because each right sibling can be computed as the difference of the parent and left sibling. Further, the value of a non-root node is always between 0 and the parent value, forming the basis of mid-short coding. The root (=total sum) must again be encoded using some universal coding scheme, unless we have preliminary knowledge of the upper bound.

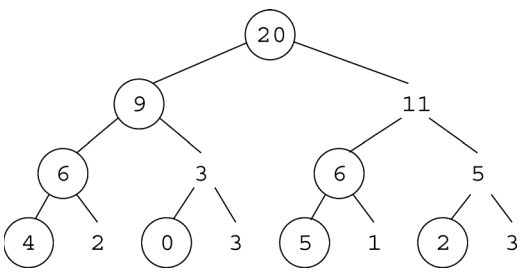

FIGURE 1. Tree-structured illustration of interpolative coding.

If n is not a power of 2, the leftmost nodes are always paired first. In [6], Moffat and Stuiver suggested non-balanced recursion as a refinement to interpolative coding, corresponding closely to the binary tree representation. Actually, this representation can lead to a possibly faster implementation of interpolative coding. The basic version has the defect that the memory references are scattered (cf. binary search), which may cause many cache misses for large input. By building an implicit binary tree of partial sums in breadth-first order, the coding can be done in a more sequential manner (cf. implicit heaps in heapsort). The details are skipped here.

As mentioned above, mid-short encoding seems a natural choice for this method, because the child is half of the parent, on the average. The example sequence, processed in breadth-first order, results in the following bounds, given in parentheses:

$$\langle 20(\infty), 9(20), 6(9), 6(11), 4(6), 0(3), 5(6), 2(5) \rangle$$

The binary code sequence, with gamma-coded root and other mid-short-coded elements is as follows (spaced for clarity).

111100101 1001 110 110 011 00 100 10

Altogether $9 + 20 = 29$ bits are needed. The number of elements should be encoded, too, but it is here excluded.

4. BASIC VERSION OF TOURNAMENT CODING

From Fig. 1 we notice that the numbers to be encoded grow when moving from lower to higher levels. For uniform distribution of leaf-level integers, each level upwards doubles the average size of numbers. In practice, doubling means an extra bit in coding. Moreover, the higher-level numbers can be arbitrarily large, which may complicate arithmetic operations.

Tournament coding also starts with pairing of numbers. Instead of their sum, the larger of the numbers is used to bound the smaller, i.e. the larger is encoded first, and then the smaller, using one of the techniques in Section 2. To encode the larger numbers, we can apply the same idea recursively, i.e. form pairs of larger numbers, of which the larger (largest of four original numbers) is encoded first, etc., recursively. A *tournament tree* illustrates the process, where the winner always becomes the parent, and the global winner is in the root, see Fig. 2. The encircled nodes are processed in breadth-first order $\langle 5, 4, 3, 3, 2, 0, 1, 2 \rangle$.

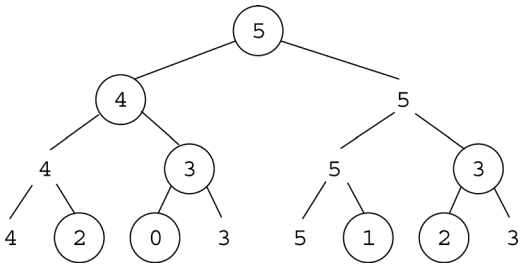

FIGURE 2. Tournament tree of an integer sequence.

Clearly, the numbers to be encoded are precisely the original integers. If some internal node is zero, all its descendants are known to be zero, and can be skipped. In other cases, we have to express the winner/loser of child pairs. A simple approach is to use one indicator bit for this information, i.e. $n - 1$ bits, altogether, for n integers. Later we shall elaborate on this detail. The smaller child is encoded using some semi-fixed-length code, bounded by the parent. For uniform distribution, high-short coding seems natural, at least on higher levels of the tree, because siblings are maxima of the same number of integers, and thus tend to be of the same order of magnitude.

The example sequence results in the following encoding. The bounds are again expressed in parentheses. The left/right indicators (*l*/*r*), denoting the smaller child, are in square brackets.

$$\langle 5(\infty), 4(5)[l], 3(4)[r], 3(5)[r], 2(4)[r], 0(3)[l], 1(5)[r], 2(3)[l]\rangle$$

The corresponding binary code sequence is as follows (with gamma-coded root, $left = 0$, $right = 1$ and bits grouped for clarity):

11010 10 0 10 1 011 1 01 1 00 0 001 1 10 0

Altogether this makes $5 + 23 = 28$ bits, which is of the same order as with interpolative coding. However, the example is too small to draw any further conclusions.

One point to note is the difference from static coding: the same source integer can get different codes in different contexts. Therefore, it is natural to compute the code values dynamically, instead of using a large number of codebooks. The same holds for decoding.

Decoding is analogous to encoding: the root is first determined by applying the related universal decoding method. The rest of the tree is then built top–down in breadth-first order, by using the known parent to bound the children, and thus determine the codebook to be applied in the semi-fixed-length decoding. The indicator bit signals whether the decoded value is the left or right child. The other child will have a value equal to the parent. If the parent is zero, this value is copied to both children without any decoding.

As for related work, the *RBUC* (Recursive Bottom–Up, Complete) coding by Moffat and Anh [20, 21] shares the idea of recursive encoding of maxima. However, it is applied to the so-called selectors (expressing codeword lengths), not the actual source integers. The source is processed in groups of s numbers, and their maximal selector is common for the whole group. Non-maximal selectors are not encoded at all. The maximal selectors of the first level are processed recursively. The method is some kind of hybrid between gamma coding and fixed-length binary coding, and essentially different from tournament coding.

5. ELEMENTARY ANALYSIS

We analyse here the case where n source integers are independent and uniformly distributed within $[0, 2k]$, with mean k. The non-uniform cases are studied experimentally in Section 7. We first derive an estimate $N_{\mathrm{IC}}(k, n)$ for the expected number of bits in interpolative coding by using the real-valued information amounts of bounded codes for integers. The leaf-level nodes are bounded by sums of two integers, with average sum $= 2k$. Thus we need $\log_2(2k + 1)$ bits, in an average, for each leaf. The nodes on the next level are double as large, taking $\log_2(4k + 1)$ bits, and so on. The total expected number N_{IC} of bits is obtained from the series

$$N_{\mathrm{IC}}(k, n) = \frac{n}{2}\log_2(2k + 1) + \frac{n}{4}\log_2(4k + 1) + \frac{n}{8}\log_2(8k + 1) + \cdots \quad (1)$$

For large k, this can be approximated as

$$N_{\mathrm{IC}}(k, n) \approx \frac{n}{2}(\log_2 k + 1) + \frac{n}{4}(\log_2 k + 2) + \frac{n}{8}(\log_2 k + 3) + \cdots = \left(\frac{n}{2} + \frac{n}{4} + \frac{n}{8}\cdots\right)\log_2 k + 0 \cdot \frac{n}{1} + 1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + \cdots = n(\log_2 k + 2) \quad (2)$$

In practical (mid-short) coding, the information amount of each element must be an integer, so the logarithms must be rounded up or down. On the leaf level, random fluctuations are expected to even out these two, but on higher levels, being sums of integers, the 'down' direction should be statistically dominating. The above bound for $N_{\mathrm{IC}}(k, n)$ is thus somewhat pessimistic.

For tournament coding, all numbers to be encoded are between 0 and $2k$, so all nodes (except the root) can be encoded with at most $\lceil \log_2(2k + 1) \rceil$ bits by using some bounded semi-fixed-length code for integers. By adding the

indicator bit and ignoring the ceiling function (as above), we get an upper bound for the computational information amount as $\approx n(\log_2 k + 2)$ bits. However, this is too pessimistic, because on lower levels, coding is bounded by the larger sibling that is usually smaller than $2k$. The leaf-level nodes are bounded by the larger of two, the second lowest level nodes are bounded by the largest of four, the third lowest level nodes by the largest of eight, and so on. In order to get a better estimate for the information amount, we need expectations for these largest-of-group values. Denote by $L(m, k)$ the largest of m elements distributed uniformly within $[0, 2k]$. By using real-valued (continuous) variables as an approximation of the discrete case, we simplify the calculations. In this approximation, only non-empty intervals get non-zero probabilities. All intervals of size 1 have the probability $1/2k$. According to the basic property of independent, uniformly distributed variables, we obtain

$$\text{Prob}[L(m, k) \leq x] = \left(\frac{x}{2k}\right)^m \tag{3}$$

By differentiating this, we obtain the related probability density function. The expectation of $L(m, k)$ is now computed as the integral

$$\begin{aligned} E[L(m,k)] &= \int_0^{2k} \frac{\mathrm{d}(x/2k)^m}{\mathrm{d}x} x \,\mathrm{d}x \\ &= \int_0^{2k} \frac{m}{2k}\left(\frac{x}{2k}\right)^{m-1} x \,\mathrm{d}x \\ &= 2k \cdot \frac{m}{m+1} \end{aligned} \tag{4}$$

So, for example $E[L(2, k)] = (4/3)k$, $E[L(4, k)] = (8/5)k$, $E[L(8, k)] = (16/9)k$, and so on, approaching $2k$, as expected. The estimated information amount (real-valued number of bits) of the tournament tree is a sum

$$\begin{aligned} N_{\text{TC}}(k, n) &\approx \frac{n}{2}\log_2 E[L(2,k)] + \frac{n}{4}\log_2 E[L(4,k)] + \cdots \\ &\approx \sum_{i=1}^{[\log_2 n]} \left[\frac{n}{2^i}\log_2\left(2k \cdot \frac{2^i}{2^i+1}\right)\right] + (n-1) \end{aligned} \tag{5}$$

where the last $+(n-1)$ comes from the left/right indicator bits. We try to simplify this formula by approximation. Integration would be one option, but the error involved is too big. We notice that only the first few terms are essential; then the ratio $2^i/(2^i+1)$ tends very rapidly to one, after which the sum is close to a geometric series. By computing the first five terms separately we get an upper bound

$$N_{\text{TC}}(k, n) \leq n(\log_2 k + 1.60) \tag{6}$$

Again, this is only a computational estimate for the case where a 'fractional' number of bits can be used for a codeword. In semi-fixed-length coding, rounding each codeword length up or down changes the actual number by less than one bit up or down. In practice, the roundings are expected to compensate each other for the most part.

As a conclusion from the analysis, it is reasonable to expect that for uniformly distributed independent integers, tournament coding is more efficient than interpolative coding by approximately 0.4 bits per number, unless roundings favour one method more than the other. Experiments will show that this estimate roughly holds for large k.

6. ELABORATIONS

Before going to experiments, we make a slight improvement to the tournament coding method. There is namely redundancy in the left/right indicator bit. Given an upper bound u for coding the smaller of two siblings, we have till now ignored the case where the siblings are equal and reserved separate codings for this single case. This defect can be removed by coupling the indicator bit with the code bits of the smaller integer. If the left child is strictly smaller, it is combined with the indicator bit according to the formula $2 \times \text{left} + 1$, giving integers $1, 3, 5, \ldots, 2u - 1$. If the right child is smaller or equal, it is combined with the indicator bit by the formula $2 \times \text{right}$, giving integers $0, 2, 4, \ldots, 2u$. There are now altogether $2u + 1$ alternative cases to be encoded (instead of the earlier $2u + 2$). This change does not essentially complicate the basic coding but yet gives notable improvement for small u.

The other elaboration relates to selection of the version of semi-fixed-length coding. We adopt the idea suggested by Moffat and Stuiver [6], namely the usage of different codes for leaves and non-leaves. In our case, we found it advantageous to use low-short coding for leaves in all cases. As concluded earlier, for non-leaf nodes high-short coding is reasonable, at least for uniformly distributed numbers, and experiments also confirm this. However, for very skew distributions, the smaller and larger of two numbers are often far apart, suggesting the mid-short or even low-short variant. One of our experiments will exemplify this.

7. EXPERIMENTAL RESULTS

In the first experiment, we created 100 000 independent, uniformly distributed integers within $[0, 2k]$. However, the encoder is not supposed to have prior knowledge about the distribution, nor the value of k. In Table 2, the bits per number are reported for five non-statistical coding methods, of which tournament coding is the clear winner, and interpolative coding is the second best. The difference of bits per number

TABLE 2. Experimental results (bits per number) of coding uniformly distributed numbers within [0, $2k$].

$2k$	Gamma	Delta	Fibonacci	Interpolative	Tournament	Entropy
1	2.000	2.500	2.500	1.577	1.218	1.000
2	2.332	2.998	2.999	2.259	1.940	1.585
4	3.399	3.799	3.599	3.085	2.762	2.322
8	4.553	4.998	4.443	3.976	3.650	3.170
16	5.939	6.527	5.470	4.924	4.578	4.087
32	7.543	7.786	6.605	5.897	5.532	5.044
64	9.306	8.906	7.952	6.883	6.504	6.022
128	11.169	9.991	9.285	7.877	7.488	7.010

TABLE 3. Experimental results (bits per number) of coding exponentially distributed numbers for different bases.

Base	Gamma	Delta	Fibonacci	Interpolative	Tournament	Entropy
3.0	1.744	2.041	2.458	1.476	1.562	1.379
2.0	2.266	2.649	2.821	2.142	2.197	2.000
1.5	3.049	3.475	3.376	2.939	2.965	2.756
1.25	4.117	4.579	4.141	3.826	3.829	3.610
1.125	5.455	5.883	5.103	4.767	4.752	4.530
1.0625	7.015	7.199	6.226	5.737	5.710	5.487

for these two methods approaches 0.4 when k increases. This matches closely with the approximative analysis of Section 5.

Note that although the results are obtained with the improved version of tournament coding, described in the previous section, and the analysis was for the basic version, the improvement tends to zero when k increases. For $2k = 1$ the advantage was 0.5 bits, but for $2k = 128$ only 0.016 bits per integer.

The values reported in Tables 2 and 3 are averages of 10 runs. The variances among repetitions were negligible.

The second experiment involves a similar comparison for integers approximating a negative exponential distribution, in order to test the effects of skewness. The continuous cumulative distribution function is

$$F(x) = 1 - \mathrm{e}^{-\lambda x} \tag{7}$$

The actual base of exponentiation is thus $b = \mathrm{e}^{\lambda}$. The numbers are generated from

$$r_{\mathrm{exp}} = \left\lfloor -\frac{\log r}{\log b} \right\rfloor \tag{8}$$

where r represents random real numbers from (0, 1). The results are reported in Table 3, and show a somewhat different behaviour: Now interpolative coding is the best for larger base values (i.e. for skewer distributions), and tournament coding for smaller values. Actually, the smaller values represent distributions closer to a uniform distribution, and thus confirm the observation made in Table 2.

We now turn to more practical cases, to find the situations where tournament coding can be recommended. We start with text compression, where one of the popular techniques is based on the Burrows–Wheeler transform [7], mentioned in the Introduction. The transform does a seemingly odd thing, namely it sorts all the *rotations* of the original sequence. The last characters of the sorted rotations constitute the result of transform, together with the order number of the original sequence among the rotations. The transformed sequence contains all the original characters, but clustered according to similar contexts. A common example used in the literature is the word 'MISSISSIPPI', which transforms into 'PSSMIPISSII', plus the order number 5 of the original sequence. The reverse transform is skipped here.

The clusteredness of the transformed sequence is utilized by another transform, called the MTF coding. An MTF list Z is initialized in the example case by the alphabet <I, M, P, S>. The sequence is processed one character at a time, so that the Z-index of the current character is the transformed value. Moreover, the character is moved to the front of Z. This has the effect that subsequent occurrences of the same character map to zeroes. More generally, small indices tend to dominate, making the index sequence easily compressible. Using the given initial list Z, the MTF transform of 'PSSMIPISSII' would produce <2, 3, 0, 3, 3, 3, 1, 3, 0, 1, 0>. The example is too small to reveal the abundance of zeroes and other small indices, common in practical cases.

TABLE 4. Experimental results (bits per character) of applying various coders to MTF indices in block-sorting compression.

File	Size	Gamma	Fibonacci	Interpolative	Tournament	Tournament2	Entropy	bzip2
bib	111261	2.440	2.984	2.081	2.154	2.108	2.285	1.975
book1	768771	2.860	3.260	2.519	2.519	2.530	2.759	2.421
book2	610856	2.491	3.010	2.145	2.196	2.170	2.396	2.062
geo	102400	6.335	5.781	4.641	4.550	4.606	5.351	4.447
news	377109	2.897	3.305	2.641	2.761	2.700	2.801	2.516
obj1	21504	4.767	4.666	4.158	4.286	4.202	4.244	4.013
obj2	246814	3.307	3.424	2.724	2.882	2.780	2.755	2.478
paper1	53161	2.778	3.217	2.596	2.729	2.656	2.687	2.492
paper2	82199	2.790	3.221	2.532	2.615	2.573	2.703	2.437
pic	513216	1.650	2.444	0.964	1.023	0.976	1.206	0.776
progc	39611	2.806	3.236	2.644	2.799	2.712	2.692	2.533
progl	71646	2.106	2.753	1.835	1.960	1.885	1.906	1.740
progp	49379	2.082	2.736	1.835	1.976	1.895	1.865	1.735
trans	93695	1.931	2.638	1.688	1.842	1.745	1.629	1.528
Average	224402	2.926	3.334	2.500	2.592	2.538	2.663	2.368

The sequence of MTF indices is naturally an interesting test case for us. Table 4 shows compression results (bits per character) for the Calgary corpus [1][1], which is by far the most used test collection for compression of text-type data.

Our implementation of MTF coding did not include any prior enhancements. Especially, we did not apply run-length coding to runs of zeroes, which is known to promote some coding techniques. We wanted to show that interpolative and tournament coding are able to handle the runs effectively by themselves. On the other hand, gamma and Fibonacci coding suffer from the 0-runs. However, even preceded by a run-length step, they would not reach the compression power of the other two. This can be deduced from Fenwick's experiments [22].

In this application, interpolative coding is almost systematically better than tournament coding, the exceptions being 'book1' and 'geo'. The MTF numbers are quite far from uniform distribution, and thus confirm the observation made for exponential distribution. Also note that there are local dependencies between the numbers to be encoded. For example, zeroes tend to be clustered. This explains the phenomenon which seems to contradict information theory: both interpolative and tournament coding get below the (zero-order) entropy in many cases. This also means that non-adaptive versions of Huffman or arithmetic coding would not be competitive. The properties of Burrows–Wheeler MTF sequences are well known (see [8, 9, 22]), and the public implementations are carefully optimized.

We also tested another version of tournament coding (called Tournament2 in Table 4), where low-short coding was applied to all nodes, to favour the skew cases. The numbers are now much closer to those of interpolative coding. However, we still see a challenge of better tuning the tournament coding for skew distributions.

Table 4 also shows compression results from the commonly used *bzip2* implementation of Burrows–Wheeler compression, developed by Julian Seward.[2] This program is optimized and fine-tuned in several ways, explaining the good results. For example, it applies run-length coding to MTF numbers (mostly zeroes in practice). It also uses up to six Huffman tables for local adaptation, together with special coding of the (canonical) Huffman tables. However, our purpose is not to compete with standard text compression tools, only to compare the behaviour of various non-statistical, non-parametric coding techniques.

Our second practical experiment concerns lossless compression of digital images. Here we restrict the testing to grey-scale images (with grey levels in 0,. . ., 255), and apply a typical technique, where pixel values are predicted from the already coded neighbours. Figure 3 shows such a neighbourhood for a non-border pixel. For example, lossless JPEG includes various alternative predictions based on W_P, N_P and NW_P (West, North and Northwest) neighbours of the current pixel P. Here we apply the prediction

$$P' = \left\lfloor \frac{W_P + \mathrm{NE}_P + 1}{2} \right\rfloor \tag{9}$$

[1]At the time of writing this article, the corpus was obtainable from the address ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus.

[2]See website http://www.bzip.org.

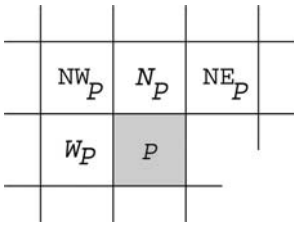

FIGURE 3. Neighbours used in predictive image compression.

which, according to our observation, gives good predictions. For the top pixel row and leftmost/rightmost columns, we use available neighbours (N_P, W_P or both). The top-left pixel is predicted to have grey-value 127. The value $d = P - P'$ can be negative, so we apply the transformation: if $d \geq 0$ then $d' = 2d$ else $d' = -2d - 1$. The obtained numbers were encoded using the same coding methods as before. The results (bits per pixel) for seven standard test images of size 512×512 are shown in Table 5. We observe that tournament coding is systematically better than the others, though not as much as for uniform distribution.

These kinds of prediction errors are known to follow roughly a (two-sided) geometric distribution, for which Golomb coding [3] is known to be nearly optimal. However, Golomb code is *parametric*, requiring some knowledge of the source properties. The compression results for Golomb coding are included in Table 5, but even though optimal parameter values were used for each case, Golomb coding does not reach the compression rates of tournament coding. The reason is that Colomb code assumes independence of source numbers, which does not quite hold, because prediction succeeds better in low-energy regions. Interpolative and tournament coding, instead, adapt themselves automatically to these variations. For the same reason these two get, in some cases, below the 0-order entropy; tournament coding actually for all but one source image.

The combined results of the compression effectiveness of interpolative and tournament coding are gathered in Fig. 4, relative to the 0-order entropy. In order to get some idea of the general trends, the regression lines are included, showing slightly growing gain for tournament coding for high-entropy sources.

We have not yet discussed the speed of tournament coding. Due to its simplicity, the method is quite fast. We wrote a straightforward C implementation and made experiments using a double-processor 1.8 GHz Intel Xeon. Table 6 shows some encoding and decoding times for interpolative and tournament coding. The source sequences (length n) consisted of 1-byte integers. The times are averages of $10^9/n$ runs, with I/O-times excluded. The growth of execution times is close to linear, as it should be.

For comparison, bzip2 compression took 0.66 s for one million bytes, and decompression (program bunzip2) 0.32 s. The comparison is naturally not fair because bzip2 includes also the modelling steps. However, the claim is that tournament coding is fast enough to be used as the final 'entropy coder' of a full-scale data compression program. Interpolative coding and decoding were slightly slower for large n because mid-short and mid-long coding routines are a bit more time-consuming than high-short and low-short, used in tournament coding.

Both implementations used explicit binary trees (see Figs 1 and 2), so that the memory consumption was of the order $2n$. The advantage was the ability to process the data in breadth-first, i.e. sequential order, which is beneficial for cache usage.

8. CONCLUSIONS

We have introduced a new non-statistical, non-parametric source coding technique for non-negative integers. It is based on recursive 'tournaments', i.e. selections of maxima, to restrict the ranges of integers, and thus laying the basis for semi-fixed-length coding. The method was inspired by interpolative coding; both are simple, fast, effective and able to take advantage of local dependencies between the numbers, such as clusters of zeroes and non-zeroes.

Both interpolative and tournament coding are basically off-line methods, so that the whole source sequence is needed at the start. This is not usually a serious restriction

TABLE 5. Experimental results (bits per pixel) from lossless coding of prediction errors for greyscale images.

Image	Gamma	Fibonacci	Interpolative	Tournament	Golomb	Entropy
Barbara	6.861	6.114	5.441	5.251	5.914	5.672
Boat	5.930	5.445	4.916	4.783	5.132	5.015
Clown	5.715	5.296	4.784	4.606	4.850	4.778
Goldhill	6.352	5.752	5.207	5.057	5.165	5.130
Lena	5.938	5.452	4.967	4.844	4.939	4.869
Mandrill	7.927	6.884	6.122	5.943	6.112	6.073
Peppers	6.284	5.698	5.176	5.036	5.039	4.998
Average	6.430	5.806	5.230	5.074	5.307	5.219

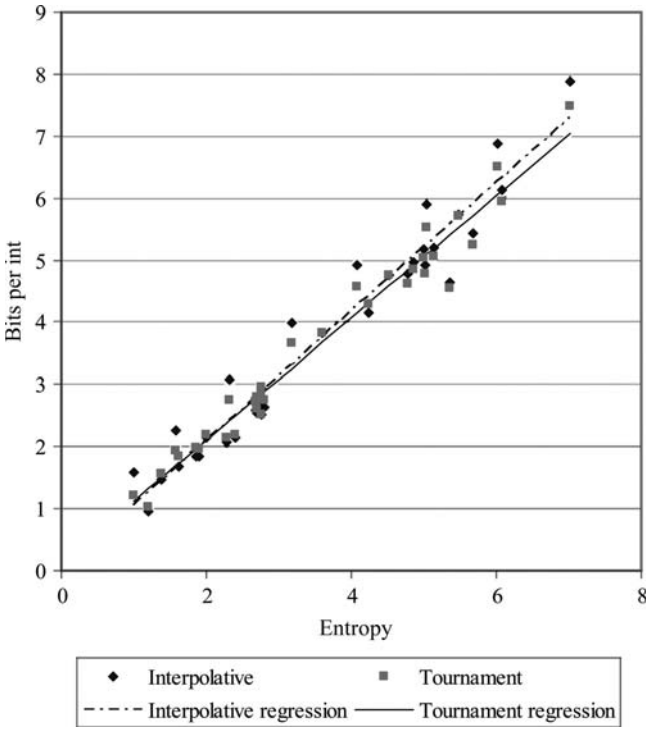

FIGURE 4. Combined results from experiments for interpolative and tournament coding.

TABLE 6. Execution times (s) of encoding and decoding integers of size 0, . . . , 128.

	Encoding time		Decoding time	
n	Interpolative	Tournament	Interpolative	Tournament
10 000	0.0005	0.0005	0.0006	0.0005
100 000	0.0059	0.0053	0.0060	0.0053
1 000 000	0.0640	0.0557	0.0604	0.0541

because the source can be processed in blocks of suitable size, as done in quite many compression packages.

Tournament coding was shown to be superior to interpolative coding for uniformly distributed (high-entropy) numbers. For skew distributions, interpolative coding was slightly better in our experiments. Other non-statistical coding techniques are usually far behind these two, except if the distribution of numbers happens to match with the model behind the method, and if the subsequent numbers to be encoded are independent.

Tournament coding can be used either as a self-dependent compression method, if the source is directly a sequence of integers, or more probably, as the final coding module of a compression package when statistical methods are either not applicable, too slow, or avoided for patent reasons.

Our future work includes more effective combination of various semi-fixed-length codes, in order to improve the compression performance for skew distributions. More experimentation is also needed with various sources. For example, it would be interesting to embed tournament coding in current lossy image and video compression methods.

REFERENCES

[1] Bell, T.C., Witten, I.H. and Cleary, J.G. (1989) Modeling for text compression. *ACM Comput. Surv.*, **21**, 557–591.

[2] Bell, T.C., Cleary, J.G. and Witten, I.H. (1990) *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.

[3] Golomb, S.W. (1966) Run-length encodings, *IEEE Trans. Inform. Theory*, **12**, 399–401.

[4] Pennebaker, W.B. and Mitchell, J.L. (1993) *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, NY.

[5] Moffat, A. and Stuiver, L. (1996) Exploiting Clustering in Inverted File Compression. *Proc. 6th Data Compression Conf.*, Snowbird, Utah, March 31–April 3 pp. 82–91, IEEE Computer Society Press, Los Alamitos, CA.

[6] Moffat, A. and Stuiver, L. (2000) Binary interpolative coding for effective index compression. *Inf. Retr.*, **3**, 25–47.

[7] Burrows, M. and Wheeler, D.J. (1994) A Block-Sorting Lossless Data Compression Algorithm, Research Report 124, Digital Systems Research Center, Palo Alto, CA.

[8] Fenwick, P.M. (1996) The Burrows–Wheeler transform for block sorting text compression: principles and improvements. *Comput. J.*, **39**, 731–740.

[9] Fenwick, P.M. (2003) Burrows–Wheeler Compression. In Sayood, K. (ed.), *Lossless Compression Handbook*. Academic Press, San Diego, CA.

[10] Huffman, D.A. (1952) A method for the construction of minimum redundancy codes. *Proc. IRE*, **40**, 1098–1101.

[11] Pasco, R. (1976) Source coding algorithms for fast data compression. PhD Thesis, Stanford University, CA.

[12] Rissanen, J.J. (1976) Generalized Kraft inequality and arithmetic coding. *IBM J. Res. Dev.*, **20**, 198–203.

[13] Elias, P. (1975) Universal codeword sets and representations of the integers. *IEEE Trans. Inform. Theory*, **21**, 194–203.

[14] Apostolico, A. and Fraenkel, A. (1987) Robust transmission of unbounded strings using Fibonacci representations. *IEEE Trans. Inform. Theory*, **33**, 238–245.

[15] Fraenkel, A.S. and Klein, S.T. (1996) Robust universal complete codes for transmission and compression. *Discrete Appl. Math.*, **64**, 31–55.

[16] Teuhola, J. (1978) A compression method for clustered bit-vectors. *Inform. Process. Lett.*, **7**, 308–311.

[17] Fenwick, P.M. (2003) Universal Codes. In Sayood, K. (ed.), *Lossless Compression Handbook*. Academic Press, San Diego, CA.

[18] Brisaboa, N.R., Fariña, A., Navarro, G. and Esteller, M.F. (2003) (S, C)-Dense Coding: An Optimized Compression Code for Natural Language Databases. *Proc. String Processing and*

Information Retrieval, Manaus, Brazil, October 8–19, LNCS, Vol. 2857. Springer, Berlin, pp. 122–136,

[19] Anh, V.N. and Moffat, A. (2006) Improved word-aligned binary compression for text indexing. *IEEE Trans. Knowl. Data Eng.*, **18**, 857–861.

[20] Moffat, A. and Anh, V.N. (2005) Binary Codes for Non-Uniform Sources. *Proc. 15th Data Compression Conf.*, Snowbird, Utah, March 29–31, pp. 133–142, IEEE Computer Society Press, Los Alamitos, CA.

[21] Moffat, A. and Anh, V.N. (2006) Binary codes for locally homogeneous sequences. *Inform. Process. Lett.*, **99**, 175–180.

[22] Fenwick, P.M. (2002) Burrows–Wheeler compression with variable-length integer codes. *Softw. Pract. Exp.*, **32**, 1307–1316.