Practical Binary Adaptive Block Coder

Yuriy A. Reznik
QUALCOMM Incorporated
5775 Morehouse Drive, San Diego, CA 92121
E-mail: yreznik@ieee.org

Abstract

This paper describes design of a low-complexity algorithm for adaptive encoding/decoding of binary sequences produced by memoryless sources. The algorithm implements universal block codes constructed for a set of contexts identified by the numbers of non-zero bits in previously observed bits in a sequence. We derive a precise formula for asymptotic redundancy of such codes, which refines previous well-known estimate by Krichevsky and Trofimov [24], and provide experimental verification of this result. In our experimental study we also compare our implementation with existing binary adaptive encoders, such as JBIG's Q-coder [41], and MPEG AVC (ITU-T H.264)'s CABAC [43] algorithms.

1 Introduction

One of the most basic tasks in the design of today's data compression algorithms is the one of converting input sequences of bits with some unknown distribution into a decodable bitstream. This happens, for example, in the design of image or video codecs, scalable (bit-slice based) encoding of spectrum in audio codecs, etc. In most such cases, the bits to be encoded are taken from values produced by various signal processing tools (transforms, prediction filters, etc), which means that they are already well de-correlated, and that assumption of memorylessness of such a source is justified.

Most commonly, the problem of encoding of such sequences of bits is solved by using fast (typically multiplication-free) approximations of binary adaptive arithmetic codes. Two well known examples of such algorithms are IBM's Q-coder [41] adopted in JBIG image coding standard [42], and CABAC encoder [43] used in MPEG AVC/ITU-T H.264 standards for video compession [44].

In this paper we describe an alternative implementation of adaptive encoder using an array of Huffman codes designed for several estimated densities, indexed by the numbers of non-zero bits in previous blocks (contexts) in a sequence.

We study both efficiency and implementation aspects of such a scheme and show that by using even relatively short blocks (8...16 bits) it can achieve compression performance comparable or superior to one of the above quoted algorithms.

This paper is organized as follows. In Section 2 we provide background information about our coding problem. In Section 3 we quote known results about efficiency of such codes and offer a more precise result. In Sections 4 and 5 we describe design of our system, and in Section 6 we provide experimental results. Appendix A

contains proofs of out Theorem 1, and Appendix B contains complete code of the program we've designed.

2 Background Information

Consider a memoryless source producing symbols from a binary alphabet $\{0,1\}$ with probabilities p, and $q = 1 - p$ correspondingly. If w is a word of length n produced by this source, then its probability:

$$\Pr(w) = p^k q^{n-k}, \tag{1}$$

where k denotes the number of 1's in this word (sometimes k is also referred to as *weight* of w).

A *block code* ϕ is an injective mapping between words w of length $|w| = n$ and binary sequences (or *codewords*) $\phi(w)$:

$$\phi : \{0,1\}^n \to \{0,1\}^* , \tag{2}$$

where the codewords $\phi(w)$ represent a *uniquely decodable* set [7].

Typically, when the source (i.e. its probability p) is known, such a code is designed to minimize its average length, or (in relative terms) its *average redundancy*:

$$R_\phi(n,p) = \frac{1}{n} \sum_{|w|=n} \Pr(w)\,|\phi(w)| - H(p). \tag{3}$$

As customary by $H(p) = -p\log p - q\log q$ we denote the *entropy* of the source [7].

Classical examples of codes and algorithms suggested for solving this problem include Huffman [18], Shannon [34], Shannon-Fano [12], Gilbert-Moore [17] codes and their variants [1]. Performance of such codes is well studied, see, e.g. [16], [25], [35], [36], [30]. Analysis of their complexity can be found in [38], [39]. Many useful practical implementation techniques for such codes have also been reported, see, e.g. [1], [27],[4].

When the source is not known, the best option available is to design a *universal code* ϕ^* that minimize the worst case redundancy for a class of sources [13, 8, 25]:

$$R_{\phi^*}(n) = \inf_{\phi} \sup_{p} R_\phi(n,p) .$$

An example of such a code can be constructed using the following estimates of words' probabilities[1]:

$$P_{KT}(w) = \frac{\Gamma(k+1/2)\,\Gamma(n-k+1/2)}{\pi\Gamma(n+1)}, \tag{4}$$

where $\Gamma(x)$ is a Γ-function, k is the weight of word w, and n is its length.

Finally, we might be in a situation when exact value of parameter of the source is not known, but we can access a sequence of symbols u produced by this source in the past. We will call such a sequence a *sample*, and will assume that it is $|u| = t$

[1]This formula is due to Krichevsky and Trofimov [24], and it assures uniform (in p) convergence to true probabilities with $n \to \infty$. See [26] and [40] for discussions on its background and optimality.

bits long. The task here is to design a *set of codes* (indexed by different values of this sample) ϕ_u^*, such that their resulting *worst case average redundancy* is minimal:

$$R_{\phi_u^*}(n,t) = \inf_{\{\phi_u\}} \sup_p \sum_{|u|=t} \Pr(u) R_{\phi_u}(n,p)\,. \tag{5}$$

Such codes are called *sample-based* or *adaptive* universal block codes [22, 23, 24].

In this paper we will study a particular implementation of adaptive block codes utilizing the following estimates of probabilities of words w given a sample u:

$$P_{KT}(w|u) = \frac{P_{KT}(u\,w)}{P_{KT}(u)} = \frac{\Gamma(k+s+1/2)\,\Gamma(n+t-k-s+1/2)}{\Gamma(s+1/2)\,\Gamma(t-s+1/2)}\,\frac{\Gamma(t+1)}{\Gamma(n+1)}\,, \tag{6}$$

where s is the weight of a sample u, and t is its length.

3 Performance of Adaptive Block Codes

The idea and original analysis of sample-based codes utilizing estimator (6) belong to R. E. Krichevsky [23]. In particular, he has shown (cf. [24, Theorem 1], [25, Theorem 3.4.1]), that the average redundancy rate of an adaptive block code is asymptotically

$$R_{\phi_u^*}(n,t) \sim \frac{1}{2\,n}\log\frac{n+t}{t}\,, \tag{7}$$

where n is a block size, and t is the size of samples.

From (7) it follows that by using samples of length $t = O(n)$ it is possible to lower redundancy rate of such codes to $O\left(\frac{1}{n}\right)$, which matches the order of redundancy rate of block codes for known sources [7, 25, 36].

More precise analysis of adaptive block codes was recently performed by Reznik and Szpankowski [29], leading to the following result.

Theorem 1 (Reznik & Szpankowsky 2003) *The average redundancy rate of an adaptive block code ϕ_u^* has the following asymptotic behavior ($n, t \to \infty$):*

$$\begin{aligned} R_{\phi_u^*}(n,t,p) &= \sum_{|u|=t} \Pr(u) R_{\phi_u^*}(n,p) \\ &= \frac{1}{n}\left\{\frac{1}{2}\log\frac{t+n}{t} + \Delta_{\phi_u^*}(n,t,p) + \frac{1-4\,p\,q}{24\,pq}\,\frac{n}{t\,(t+n)} - \frac{1-3\,pq}{24\,p^2q^2}\,\frac{(n+2\,t)\,n}{t^2\,(t+n)^2}\right. \\ &\qquad \left. + O\left(\frac{1}{t^3}+\frac{1}{n^3}\right)\right\}\,, \end{aligned} \tag{8}$$

where n is a block size, and t is a sample size, $p, q = 1-p$ are probabilities of symbols of the input source, and where

$$\Delta_{\phi_u^*}(n,t,p) = \sum_{|u|=t}\sum_{|w|=n} \Pr(u)\Pr(w)\left[|\phi_u^*(w)| + \log P_{KT}(w|u)\right] \tag{9}$$

is the average redundancy of code ϕ_u^* with respect to estimated distribution (6).

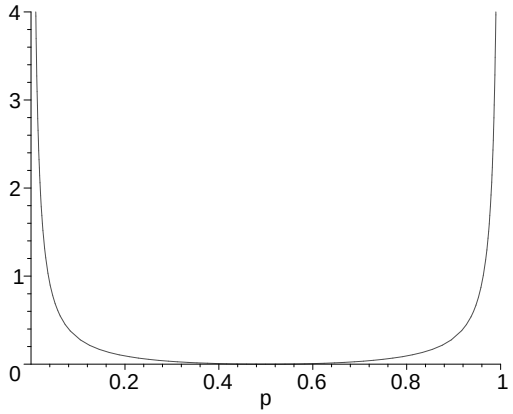

Figure 1: Behavior of a factor $\frac{1-4\,p\,q}{24\,pq}$ in redundancy expression (8).

The exact behavior of $\Delta_{\phi_u^*}(n,t,p)$ is algorithm-specific, but for most conventional coding techniques, including Huffman and Shannon codes, this term can be bounded as follows:

$$|\Delta(n,t,S)| \leqslant 1\,.$$

We also notice, that the redundancy of such codes also becomes affected by the following term:

$$\frac{1-4\,p\,q}{24\,pq}\,\frac{n}{t\,(t+n)}$$

which is a function of the parameter of the source p. We plot leading factor of this term in Figure 1, and conclude that for short blocks/samples performance of such codes becomes sensitive to the asymmetry of the source.

Proof of Theorem 1 can be found in Appendix A.

4 Efficient Implementation of Block Codes

We first notice, that in a memoryless model the probability of a word w (or its estimate, cf. (1), (4), (6)) depends only on its weight k, but not an actual pattern of its bits. Hence, considering a set of all possible n-bit words, we can split it in $n+1$ groups:

$$\{0,1\}^n = W_{n,0} \cup W_{n,1} \cup \ldots \cup W_{n,k} \cup \ldots \cup W_{n,n}, \tag{10}$$

containing words of the same weight ($k = 0,\ldots,n$), and the same probability. As obvious, the sizes of such groups are $|W_{n,k}| = \binom{n}{k}$. For further convenience, we will assume that each group $W_{n,k}$ stores words in a lexicographic order. By $I_{n,k}(w)$ we will denote the index (position) of a word w in a group $W_{n,k}$.

To describe the structure of our proposed mapping between words in groups $W_{n,k}$ and their codewords, we will use an example code shown in Table 1. This code was constructed using a modification of Huffman's algorithm [18], in which additional steps were taken to ensure that codewords located at the same level have same lexicographic order as input blocks that they represent. It is well-known that such a reordering is possible without loss of compression efficiency. Examples of prior

Table 1: Example of a code constructed for 4-bit blocks with Bernoulli probabilities: $p^k q^{n-k}$, $p = 0.9$.

Block w	k	$I_{n,k}(w)$	$\Pr(w)$	**Length**	**Code** $\phi(w)$	**Sub-group**
0000	0	0	0.6561	1	1	0
0001	1	0	0.0729	3	001	1
0010	1	1	0.0729	3	010	1
0011	2	0	0.0081	6	000011	3
0100	1	2	0.0729	3	011	1
0101	2	1	0.0081	7	0000001	4
0110	2	2	0.0081	7	0000010	4
0111	3	0	0.0009	9	000000001	5
1000	1	3	0.0729	4	0001	2
1001	2	3	0.0081	7	0000011	4
1010	2	4	0.0081	7	0000100	4
1011	3	1	0.0009	9	000000010	5
1100	2	5	0.0081	7	0000101	4
1101	3	2	0.0009	9	000000011	5
1110	3	3	0.0009	10	0000000001	6
1111	4	0	0.0001	10	0000000000	7

algorithms using reordering to simplify designs include "Huffman-Shannon-Fano" codes [5], "canonic Huffman codes" of Moffat and Turpin [27], [4], etc.

In Figure 2 we depict the structure of this code. As expected, each group $W_{n,k}$ consists of at most two *sub-groups* containing codewords of the same length: [2]

$$W_{n,k} = W_{n,k,\ell} \cup W_{n,k,\ell+1}\,, \tag{11}$$

where ℓ is the shortest code length that can be assigned to blocks from $W_{n,k}$. Moreover, since words within $W_{n,k}$ group follow lexicographic order, then the split between $W_{n,k,\ell}$ and $W_{n,k,\ell+1}$ is simply:

$$W_{n,k,\ell} = \{w \in W_{n,k} : I_{n,k}(w) < n_k\}\,, \tag{12}$$

$$W_{n,k,\ell+1} = \{w \in W_{n,k} : I_{n,k}(w) \geqslant n_k\}\,, \tag{13}$$

where n_k denotes the size of a subgroup with shorter codewords.

We will call lexicographically smallest codewords in each subgroup *base codewords*:

$$B_{n,k,\ell} = \phi(w_0)\,, \tag{14}$$

$$B_{n,k,\ell+1} = \phi(w_{n_k})\,, \tag{15}$$

where w_i : is i-th block in $W_{n,k}$, and note that the remaining codewords in both subgroups can be computed as follows:

$$\phi(w_i) = \begin{cases} B_{n,k,\ell} + i, & \text{if } i < n_k\,, \\ B_{n,k,\ell+1} + i - n_k, & \text{if } i \geqslant n_k\,. \end{cases} \tag{16}$$

[2] This follows from the fact that all words in $W_{n,k}$ have the same probability, and so-called *sibling property* of Huffman codes (cf. [16], [35], [36]). This observation also holds true for Generalized Shannon codes [10] and possibly some other algorithms.

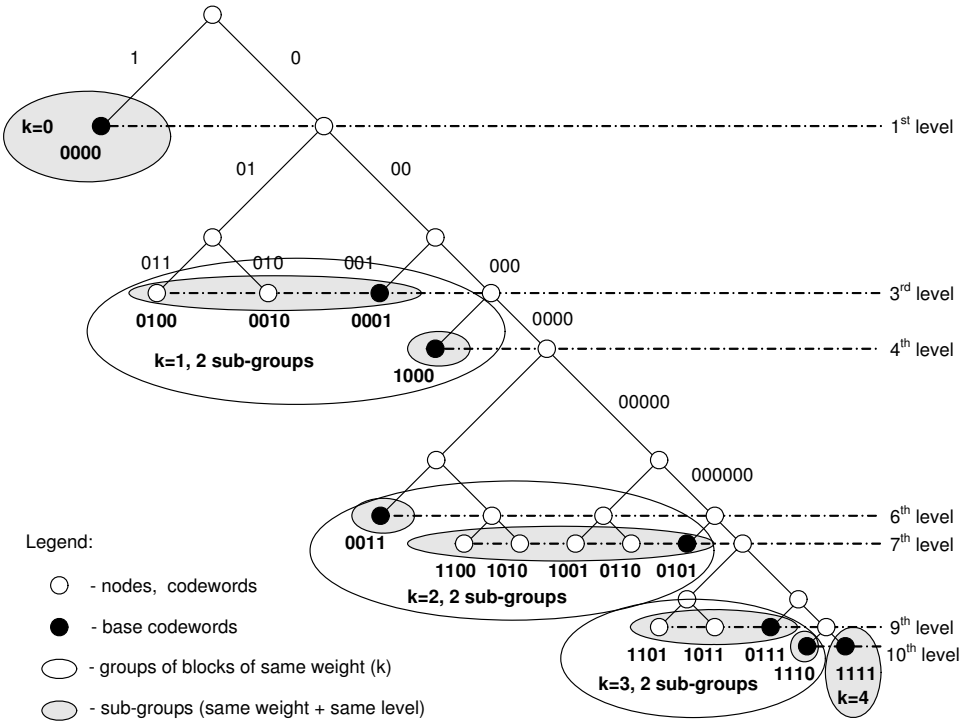

Figure 2: Structure of an example block code.

We point out that such base codewords are only defined for non-empty subgroups, and that the number of such subgroups S in a tree constructed for n-bit blocks is within:

$$n + 1 \leqslant S \leqslant 2\,n\,. \tag{17}$$

We also notice that multiple subgroups can reside on the same level [3] (see e.g. level 10 in tree in Figure 2), and the number of such collocated sub-groups cannot be greater than $n + 1$.

4.1 Proposed Algorithm for Block Encoding/Decoding

Based on the discussion above we can now define a simple algorithm for direct computation of block codes.

We assume that parameters n_k $(0 \leqslant k \leqslant n)$ are available, and that for each non-empty sub-group we can obtain its level ℓ and its base codeword $B_{n,k,\ell}$. Then the process of encoding a block w is essentially a set of the following steps:

- using w obtain its weight k, and index $I_{n,k}(w)$
- if $I_{n,k}(w) < n_k$ use first subgroup $W_{n,k,\ell}$ otherwise pick $W_{n,k,\ell+1}$
- retrieve base codeword and compute the code according to (16).

[3] This is one of the most obvious differences between our algorithm and Connell [5], or Moffat and Turpin [27] algorithms, which assign unique base codewords for each level, but then they need an $O(n\,2^n)$-large reordering table to work with such codes. Here, the entire structure is $O(n^2)$ bits large. Also, unlike [38], [39] our algorithm does not assume any particular order of probabilities based on weight k. This way we can implement codes for universal densities (4), and (6).

Algorithm 1 Direct construction of block codes.

```
/* encoder structure: */
typedef struct {
  unsigned short  nk[N+1];        /* # of elements in first (n,k) subgroup */
  unsigned char   sg[N+1][2];     /* (k,j) -> subgroup index mapping */
  unsigned char   len[S];         /* subgroup -> code length mapping */
  unsigned int    base[S];        /* subgroup -> base codeword mapping */
} ENC;

/* block encoder: */
unsigned block_enc (unsigned w, ENC *enc, BITSTREAM *bs)
{
  unsigned i, j, k, len, code;

  k = weight(w);                  /* split w into (k,index) */
  i = index(n,k,w);
  if (i >= enc->nk[k]) {          /* find subgroup containing w */
    i -= enc->nk[k];              /* adjust index */
    j = enc->sg[k][1];
  } else
    j = enc->sg[k][0];
  code = enc->base[j] + i;        /* generate code */
  len = enc->len[j];
  put_bits(code, len, bs);        /* write code to bitstream */

  return k;
}
```

A complete C-language code of such a procedure is presented as Algorithm 1 above.

It can be seen that memory-wise this algorithm needs only S base codewords ($O(n)$-bit long[4]), $n+1$ values n_k ($O(n)$-bit long), S code lengths ($O(\log n)$-bit long), and $2\,(n+1)$ subgroup indices ($O(\log n)$-bit long). Given the fact that $S = O(n)$, the entire structure needs $O(n^2)$ bits.

In a particular implementation shown in Algorithm 1, and assuming, e.g. that $n = 20$ and $S = 32$, the size of this structure becomes 244 bytes - far less than 2^{20} words needed to present this code in a form of a direct table.

We note that for reasonably short blocks (e.g. $n \leqslant 12 \ldots 16$) computation of their weights and indices (functions `weight(.)` and `index(.,.)` in Algorithm 1), can be a matter of a single lookup, in which case, the entire encoding algorithm needs at most 1 comparison, 2 additions, and 4 lookups.

For larger blocks, one can use the following well-known combinatorial formula (cf. [28], [2], [33], [6], [38], [39]):

$$I_{n,k}(w) = \sum_{j=1}^{n} w_j \binom{n-j}{\sum_{k=j}^{n} w_k}, \tag{18}$$

[4]We note that additional memory reduction is possible by storing incremental values of base codewords – this is discussed in a companion paper [32].

where w_j represent individual bits of the word w, and it is assumed that $\binom{n}{k} = 0$ for all $k > n$. In order to implement it, one could either pre-compute all binomial coefficients up to level n in Pascal's triangle, or compute them dynamically, using the following simple identities:

$$\binom{n-k}{k-1} = \frac{k}{n}\binom{n}{k}, \quad \text{and} \quad \binom{n-k}{k} = \frac{n-k}{n}\binom{n}{k}.$$

The implementation based on pre-computed coefficients requires $\frac{n\,(n+1)}{2} = O\left(n^2\right)$ words ($O(n^3)$ bits) of memory, and $O(n)$ additions. Dynamic computation of coefficients will require $O(n)$ additions, multiplications and divisions, but the entire process needs only few registers. Additional discussion on complexity of index computation can be found in [39].

We now turn our attention to the design of a decoder. Here, we will also need parameters n_k, base codewords, and their lengths. For further convenience (as it was suggested by Moffat and Turpin [27]) we will use left-justified versions of base values:

$$B^{lj}_{n,k,\ell} = B_{n,k,\ell}\; 2^{T-\ell}, \tag{19}$$

where T is the length of a machine word ($T > \max \ell$). We will store such left-justified values in a lexicographically decreasing order. Then, the decoding process can be described as follows:

- find first (top-most) subgroup with $B^{lj}_{n,k,\ell}$ being less than last T bits in bit-stream,
- decode index of a block $I_{n,k}(w)$ (based on (16)), and
- produce reconstructed block using its weight k and index.

A complete C-language code of such a procedure is presented as Algorithm 2.

We note that (besides using left-justified base words) this algorithm has almost identical data structure. The only new elements here are weights k and subgroup level indicators j ($j = 0$ if subgroup contains shorter codewords, and $j = 1$ otherwise). Memory-wise it has very similar characteristics.

The main decoding process requires between 1 and S comparisons and lookups to find a subgroup, 1 or 2 additions, 1 shift, 1 extra comparison, and 3 extra lookups.

As in Moffat-Turpin algorithm [27] the number of steps needed for finding a subgroup can be further reduced by placing base codewords in a binary search tree or using an extra lookup table, but in both cases we need to use extra memory to accomplish this.

We note, that at the end of the decoding process we also need to convert word's weight k and index $I_{n,k}(w)$ into its actual value (function `word()` in Algorithm 2). If blocks are reasonably short, this can be accomplished by a simple lookup. Otherwise, we can synthesize the word by using the enumeration formulae (18). Complexity-wise this process is similar to index computation in the encoder.

5 Design of an Adaptive Block Coder

Using above described algorithms we can now define a system for adaptive encoding/decoding of blocks of data.

Algorithm 2 Decoding of a block codes.

```
/* decoder structure: */
typedef struct {
  unsigned short  nk[N+1];         /* # of elements in first (n,k) subgroup */
  struct {unsigned char k:7,j:1;} kj[S]; /* subgroup -> (k,j) mapping */
  unsigned char   len[S];          /* subgroup -> code length mapping */
  unsigned int    lj_base[S];      /* subgroup -> left-justified codewords */
} DEC;

/* block decoder: */
unsigned block_dec (unsigned *w, DEC *dec, BITSTREAM *bs)
{
  unsigned i, j, k, len, val;

  val = bitstream_buffer(bs);
  for (j=0; dec->lj_base[j]>val; j++) ; /* find a subgroup */
  len = dec->len[j];
  scroll_bitstream(len, bs);      /* skip decoded bits  */
  i = (val - dec->lj_base[j]) >> (32-len);
  k = dec->kj[j].k;               /* get weight */
  j = dec->kj[j].j;               /* get sub-group index */
  if (j)                          /* reconstruct index */
    i += dec->nk[k];
  *w = word(n,k,i);               /* generate i-th word in (n,k) group */

  return k;
}
```

In this system, we assume that input blocks can be encoded under the following conditions:

1. there is no context - i.e. we implement universal code,
2. the context is given by one previously seen block - i.e. $t = n$,
3. the context is given by two previously seen blocks - i.e. $t = 2\,n$.

We note, that instead of using actual blocks as contexts it is sufficient (due to memoryless nature of the source) to use their weights.

This means, that for t-bit samples, we will need to have an array of $t + 1$ code structures indexed by their weights s. To further save space, we can use symmetry of KT-distributions (6) with respect to s and k: replace $s = t - s$ and flip bits (i.e. force $k = n - k$) every time when $s > t/2$. This way we will only need to define $t/2 + 1$ tables.

Hence, the overall amount of memory needed by our adaptive code becomes $1 + n/2 + 1 + n + 1 = 1.5\,n + 3$ tables. Specific memory estimates for block sizes $n = 8 \ldots 20$, are shown in Table 2.

In out test implementation we've generated all these tables using KT-estimated densities (4) and (6), and using modified Huffman code- construction algorithm, as described in Section 3.

Table 2: Memory usage estimates [in bytes] for different block sizes

n	$\max t$	$\max S$	**Size of a single table**	**Tables for all contexts**
8	16	14	102	1530
12	24	19	140	2940
16	32	25	184	4968
20	40	29	216	7128

In Appendix B we provide a complete code of a program implementing such an encoding system.

6 Experimental Study of Performance of our Algorithm

In this section we provide experimental results of evaluation of performance of our adaptive code with block size $n = 16$, and compare it with the following well known algorithms:

- IBM's Q-coder algorithm [41] adopted in JBIG standard for image compression [42] (we've used implementation from JBIG's jbigkit);
- CABAC binary arithmetic encoder [43] from MPEG AVC/ITU-T H.264 standard for video coding [44].

In order to conduct our tests we've used computed-generated sequences of bits simulating output from a binary Bernoulli source with probability p. Lengths of such sequences ranged from 16 to 1024, and for each particular length we have generated $Q = 1000000$ samples of such sequences.

Relative redundancy rates were computed as:

$$\text{Rate} = \frac{(\text{sum of lengths of all codes produced for Q sequences})/Q - H(p)}{H(p)}$$

For our adaptive code we've used the following structure of contexts:

- first 16-bit block is encoded without context (universal code),
- second block is encoded using first one as its context (code with $t = 16$),
- third and all subsequent blocks are encoded using two previous blocks in a sequence as contexts (sample-based code with $t = 32$).

The results of our experimental study are shown in Figures 3 and 4. It can be seen that our code has a much faster rate of convergence than that of Q-coder or CABAC algorithms. It clearly outperforms them for short sequences, and becomes comparable to the best of these two when the total length of encoded bits approaches 1024.

In Figure 4 we also show analysis of sensitivity of redundancy rates of these codes to asymmetry of the source. Here, after 160 encoded bits (or 10 16-bit blocks) our algorithm delivers much lower redundancy compared to others. Its behavior is consistent with one that was predicted by our Theorem 1.

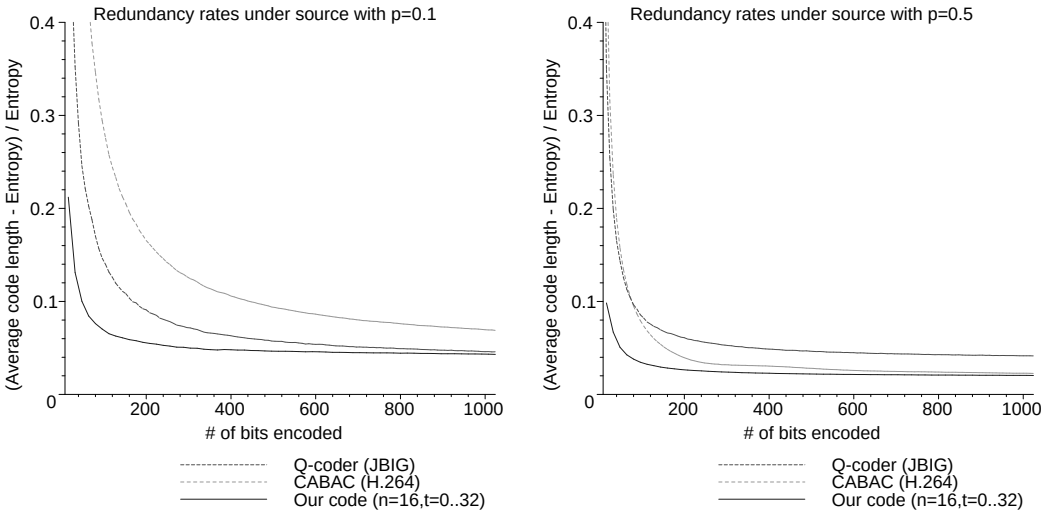

Figure 3: Comparison of redundancy rates under memoryless sources with $p = 0.1$ (left) and $p = 0.5$ (unbiased case, right).

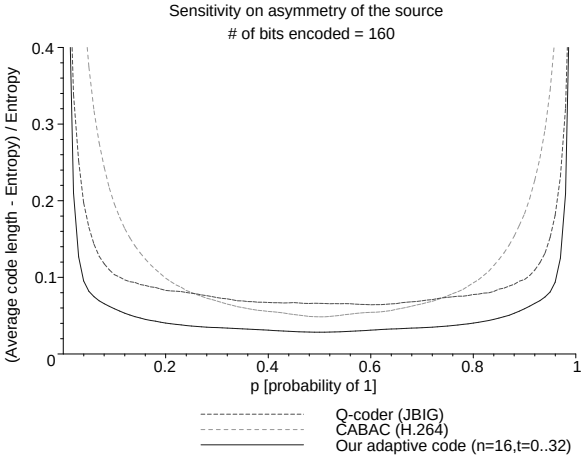

Figure 4: Sensitivity of redundancy to asymmetry of the source.

References

[1] J. Abrahams, Code and parse trees for lossless source encoding, *in Proc. Compression and Complexity of SEQUENCE'97* (Positano: IEEE Press, 1998) 145–171.

[2] V. F. Babkin, A method of universal coding with non-exponent labour consumption, *Probl. Inform. Trans.*, 1 (4) (1971) 13–21 (in Russian)

[3] A. A. Borovkov, *Probability Theory* (Gordon & Breach, 1998)

[4] A. Brodnik and S. Carlsson, Sublinear Decoding of Huffman Codes Almost in Place, *DIMACS Workshop on Codes and Trees: Algorithmic and Information Theoretic Approaches*, (October 1998, Rutgers University, DIMACS Center, NJ).

[5] J. B. Connell, A Huffman-Shannon-Fano Code, *Proc. IEEE* (July 1973) 1046–1047.

[6] T. M. Cover, Enumerative Sources Encoding, *IEEE Trans. Inform. Theory*, 19 (1) (1973) 73–77.

[7] T. M. Cover and J. M. Thomas, *Elements of Information Theory*, (John Wiley & Sons, New York, 1991).

[8] L. D. Davisson, Universal Noiseless Coding, *IEEE Trans. Inform. Theory*, 19 (6) (1973) 783–795.

[9] M. Drmota, H-K. Hwang, and W. Szpankowski, Precise Average Redundancy of an Idealized Arithmetic Coding, Proc. Data Compression Conference (2002) 222-231.

[10] M. Drmota and W. Szpankowski, Generalized Shannon Code Minimizes the Maximal Redundancy, *in Proc. LATIN'02* (Cancun, Mexico, 2002) 306–318.

[11] M. Drmota and R. Tichy, *Sequences, Discrepancies, and Applications* (Springer-Verlag, Berlin, 1997).

[12] R M. Fano, *The transmission of information.* Res. Lab. Electronics, Massachusetrs Inst. of Technology, Cambridge, Mass. Tech. Rep. *65* (1949).

[13] B. M. Fitingof, Optimal Coding in the Case of Unknown and Changing Message Statistics, *Probl. Inform. Transm.*, 2, (2) (1965) 3–11 (in Russian) 1–7 (English Transl.)

[14] P. Flajolet, Singularity analysis and asymptotics of Bernoulli sums, *Theoretical Computer Science*, 215 (1999) 371–381.

[15] P. Flajolet and A. Odlyzko, Singularity analysis of generating functions, *SIAM J. Discrete Math.*, 3 (2) (1990) 216–240.

[16] R. Gallager, Variations on the theme by Huffman, *IEEE Trans. Inform. Theory*, **IT-24** (Nov. 1978) 668–674.

[17] E. N. Gilbert and E. F. Moore, Variable-Length Binary Encodings, *Bell Syst. Tech. J.*, **7** (1959) 932–967.

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

[19] P. Jacquet, and W. Szpankowski, Entropy computations via analytic depoissonization, Tech. Report CSD-TR-96-085, Purdue University (1996)

[20] P. Jacquet, and W. Szpankowski, Analytical depoissonization and its applications, *Theoretical Computer Science*, 201 (1-2) (1998) 1–62.

[21] D. Knuth, *The Art of Computer Programming. Sorting and Searching. Vol. 3* (Addison-Wesley, Reading MA, 1973).

[22] R. E. Krichevsky, "The Connection Between Redundancy and Reliability of Information about the Source," *Probl. Inform. Trans.*, vol. 4, no. 3, pp. 48–57, 1968 (in Russian).

[23] R. E. Krichevsky, Optimal Source Coding Based on Observation, *Probl. Inform. Trans.*, 11 (1) (1975) 37–48 (in Russian).

[24] R. E. Krichevsky and V. K. Trofimov, The Performance of Universal Encoding, *IEEE Trans. Information Theory*, **27** (1981) 199–207.

[25] R. E. Krichevsky, Universal Data Compression and Retrieval. (Kluwer, Norwell, MA, 1993).

[26] R. E. Krichevskiy, Laplace's Law of Succession and Universal Encoding, *IEEE Trans. Information Theory*, **44** (1998) 296–303.

[27] A. Moffat and A. Turpin, On the Implementation of Minimum-Redundancy Prefix Codes, *IEEE Trans. Communications*, **45** (10) (1997) 1200–1207.

[28] V. I. Mudrov, An algorithm for enumeration of combinations, *Vyc. Math. and Math. Phys.*, 5 (4) (1965) 776–778 (in Russian).

[29] Y.A. Reznik and W. Szpankowski, Asymptotic average redundancy of adaptive block codes, *in Proc. Int. Symp. Information Theory (ISIT'03)* (Yokohama, Japan, 29 June-4 July 2003) 79–.

[30] Y. A. Reznik, On second-order properties of minimum redundancy block codes, *in Proc. Int. Symp. Information Theory (ISIT'04)* (Chicago, IL, 27 June-2 July 2004) 94–.

[31] Y.A.Reznik, Asymptotic properties of sample-based entropy, information divergence, and related metrics, *in Proc. Data Compression Conference (DCC'05)* (Snowbird, UT, 29-31 March, 2005) 475–.

[32] Y.A.Reznik, Memory Efficient Encoding/Decoding of Variable-Length Codes for Monotonic Distributions, *Data Compression Conference (DCC'07)* – submitted.

[33] J. P. M. Schalkwijk, An Algorithm for Source Coding, *IEEE Trans. Inform. Theory*, 18 (3) (1972) 395–399.

[34] C. E. Shannon, A mathematical theory of communication, *Bell Syst. Tech J. 27* (July 1948) 379–423.

[35] P. Stubley, On the redundancy of optimum fixed-to-variable length codes, *Proc. Data Compression Conference* (Snowbird, UT, 1994) 90-97.

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

[37] W. Szpankowski, *Average Case Analysis of Algorithms on Sequences*, (John Wiley & Sons, New York, 2001).

[38] T. Tjalkens, The complexity of minimum redundancy coding, *in Proc. IEEE Int. Symp. Inf. Theory (ISIT'00)*, (Sorrento, Italy, 2000) 373–.

[39] T. Tjalkens, Implementation cost of the Huffman-Shannon-Fano code, *in Proc. Data Compression Conference (DCC'05)* (Snowbird, Utah, March 29-31, 2005) 123–132.

[40] Q. Xie and A. R. Barron, "Minimax redundancy for the class of memoryless sources," *IEEE Trans. Inform. Theory*, vol. 43, pp. 646–657, 1997.

[41] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, Jr., R. B. Arps An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder *IBM J. Res. Dev.*, **32** (6) (1988) 717–.

[42] ITU-T Recommendation T.82.

[43] D. Marpe, H. Schwartz, and T. Wiegand. Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC video compression standard. IEEE Trans. on CSVT, 13(7):620 636, July 2003.

[44] ISO/IEC 14496-10 (MPEG-4 AVC) / ITU-T Recommendation H.264 standard for video compression.

A Proof of Theorem 1

We need to evaluate average redundancy of an adaptive code ϕ_u^* working with samples of length t and blocks of length n produced by a binary memoryless source with parameter p:

$$R_{\phi_u^*}(n,t,p) = \frac{1}{n}\sum_{|u|=t}\sum_{|w|=n}\Pr(u)\Pr(w)\,|\phi_u(w)| - H(p)\,. \tag{20}$$

We will further assume that each codeword $\phi_u(w)$ is generated on the basis of KT-estimated probability $P_{KT}(w|u)$ (6), and therefore we can rewrite (20) as follows:

$$R_{\phi_u^*}(n,t,p) = \frac{1}{n}\sum_{|u|=t}\sum_{|w|=n}\Pr(u)\Pr(w)\log P_{KT}^{-1}(w|u) - H(p) + \frac{1}{n}\Delta_{\phi_u^*}(n,t,p)\ , \tag{21}$$

where by

$$\Delta_{\phi_u^*}(n,t,p) = \sum_{|u|=t}\sum_{|w|=n}\Pr(u)\Pr(w)\left[|\phi_u(w)| + \log P_{KT}(w|u)\right] \tag{22}$$

we denote the redundancy of code ϕ_u^* with respect to the distribution it implements.

We know, that given density $P_{KT}(w|u)$ most existing minimum redundancy block codes (such as block Huffman, or Shannon algorithms) produce codewords such that:

$$\lfloor \log P_{KT}(w|u) \rfloor \leqslant |\phi_u(w)| \leqslant \lceil \log P_{KT}(w|u) \rceil\ ,$$

which implies that $\Delta_{\phi_u^*}(n,t,p)$ is a quantity of bounded magnitude:

$$\left|\Delta_{\phi_u^*}(n,t,p)\right| \leqslant 1\,,$$

and which might have some erratic or oscillating behavior (cf. [36], [9]).

We now focus our attention on the main sum in (21):

$$\begin{aligned}
&-\sum_{|u|=t}\sum_{|w|=n}\Pr(u)\Pr(w)\log P_{KT}(w|u) = \\
&\quad = -\sum_{|u\,w|=t+n}\Pr(u\,w)\log P_{KT}(u\,w) + \sum_{|u|=t}\Pr(u)\log P_{KT}(u) \\
&\quad = (t+n)\,C_{KT}(t+n,p) - t\,C_{KT}(t,p)\,,
\end{aligned} \tag{23}$$

where

$$C_{KT}(n,p) = -\frac{1}{n}\sum_{|w|=n}\Pr(w)\log P_{KT}(w)\ , \tag{24}$$

is the average rate of the KT-estimator processing n-symbols words produced by p.

A.1 Asymptotic average rates of empirical entropy and KT-estimator

Consider KT-estimated probability of a word w

$$P_{KT}(w) = \frac{\Gamma(k+1/2)\,\Gamma(n-k+1/2)}{\pi\Gamma(n+1)}\,.$$

Using Stirling's approximation (and excluding cases when $k = 0, n$), we can show that:

$$\begin{aligned} -\log P_{KT}(w) &= n\,F(w) + \frac{1}{2}\log n + \frac{1}{2}\log\frac{\pi}{2} + \frac{1}{12\,n} \\ &\quad + \frac{1}{24\,k} + \frac{1}{24\,(n-k)} + O\left(\frac{1}{k^3} + \frac{1}{(n-k)^3}\right), \end{aligned} \tag{25}$$

where:

$$F(w) = -\frac{k}{n}\log\left(\frac{k}{n}\right) - \frac{n-k}{n}\log\left(\frac{n-k}{n}\right). \tag{26}$$

is an empirical entropy [13, 24] of a word w.

The average rate of the empirical entropy $F(w)$ under source p is:

$$\begin{aligned} &\sum_{|w|=n} \Pr(w)F(w) \\ &= -\sum_{k=0}^{n}\binom{n}{k}p^k q^{n-k}\left[\frac{k}{n}\log\left(\frac{k}{n}\right) + \frac{n-k}{n}\log\left(\frac{n-k}{n}\right)\right] \\ &= \log n - \sum_{k=1}^{n}\binom{n}{k}p^k q^{n-k}\frac{k}{n}\log(k) - \sum_{k=0}^{n-1}\binom{n}{k}p^k q^{n-k}\frac{n-k}{n}\log(n-k) \\ &= \log n - p\sum_{k=1}^{n}\binom{n-1}{k-1}p^{k-1}q^{n-k}\log(k) - q\sum_{k=0}^{n-1}\binom{n-1}{k}p^k q^{n-1-k}\log(n-k) \\ &= \log n - p\,f(n-1,k,p) - q\,f(n-1,k,q). \end{aligned} \tag{27}$$

where:

$$f(n,k,\theta) = \sum_{k=0}^{n}\binom{n}{k}\theta^k(1-\theta)^{n-k}\log(1+k). \tag{28}$$

We immediately notice that (28) belongs to a class of so-called binomial sums (see, e.g. [3, p. 92]), and that for large n we must have (a uniform in θ) convergence $f(n,k,\theta) \to \log(1+\theta\,n)$. However, in order to obtain a more detailed asymptotic of $f(n,k,\theta)$, one must use analytic techniques, such as analytic depoissonization [19, 20], or singularity analysis of generating functions [15].

In fact, the last approach in application to a class of polylogarithmic Bernoulli sums has already been used by P. Flajolet [14], and in particular, he has shown that

$$\sum_{k=1}^{n}\binom{n}{k}\theta^k(1-\theta)^{n-k}\log k = \log(\theta\,n) + \frac{\theta-1}{2\,\theta\,n} - \frac{\theta^2-6\,\theta+5}{12\,\theta^2 n^2} + O\left(\frac{1}{n^3}\right),$$

which is a very similar sum to one that we need to evaluate (28). To take advantage of this existing result, we simply replace $\log(1+k)$ in (28) with:

$$\log(k+1) = \log(k) + \frac{1}{k+1} + \frac{1}{2}\frac{1}{(k+1)(k+2)} + \frac{5}{6}\frac{1}{(k+1)(k+2)(k+3)} + \ldots. \tag{29}$$

The evaluation of the sums containing factorial powers of k yields:

$$\begin{aligned} S_1(n,\theta) &= \sum_{k=1}^{n} \binom{n}{k} \theta^k (1-\theta)^{n-k} \frac{1}{k+1} \\ &= \frac{1}{\theta\,(n+1)} \left[1-(1-\theta)^n - \theta\, n\, (1-\theta)^n\right] \\ &= \frac{1}{\theta\, n} - \frac{1}{\theta\, n^2} + O\left(\frac{1}{n^3}\right), \end{aligned} \tag{30}$$

$$\begin{aligned} S_2(n,\theta) &= \sum_{k=1}^{n} \binom{n}{k} \theta^k (1-\theta)^{n-k} \frac{1}{(k+1)(k+2)} \\ &= \frac{1}{\theta^2\,(n+1)\,(n+2)} \left[1-(1-\theta)^n - \theta\, n\, (1-\theta)^n - \frac{\theta^2\, n\,(n+1)}{2}(1-\theta)^n\right] \\ &= \frac{1}{\theta^2\, n^2} + O\left(\frac{1}{n^3}\right), \end{aligned} \tag{31}$$

and it is clear that the contribution of the subsequent terms in (29) to the sum (28) is within $O\left(\frac{1}{n^3}\right)$.

Combining the above formulae, we obtain

$$f(n,k,\theta) = \log(\theta\, n) + \frac{1+\theta}{2\,\theta\, n} - \frac{\theta^2+6\,\theta-1}{12\,\theta^2 n^2} + O\left(\frac{1}{n^3}\right), \tag{32}$$

and subsequently (after plugging (32) in (27) and some simple algebra):

$$\sum_{|w|=n} \Pr(w) F(w) = H(p) - \frac{1}{2\,n} + \frac{pq-1}{12\,p\,q\,n^2} + O\left(\frac{1}{n^3}\right), \tag{33}$$

which is an (up to $O\left(\frac{1}{n^3}\right)$-term) accurate asymptotic expression for the average rate of empirical entropy.

We now focus our attention on the average rate of the KT-estimator (24). Using our asymptotic expression (25) and replacing $\frac{1}{k}$ and $\frac{1}{n-k}$ with the appropriate factorial powers we can show that:

$$\begin{aligned} C_{KT}(n,p) &= -\frac{1}{n} \sum_{|w|=n} \Pr(w) \log P_{KT}(w) \\ &= \sum_{|w|=n} \Pr(w) F(w) + \frac{1}{n} \left\{ \frac{1}{2} \log n + \frac{1}{2} \log \frac{\pi}{2} + \frac{1}{12\,n} \right. \\ &\qquad \left. + \frac{1}{24} \left[S_1(n,p) + S_2(n,p)\right] + \frac{1}{24} \left[S_1(n,q) + S_2(n,q)\right] \right\} \\ &\qquad + O\left(\frac{1}{n^4}\right), \end{aligned} \tag{34}$$

where $S_1(n,\theta)$ and $S_2(n,\theta)$ are already familiar sums (30) and (31).

Now by using (33) and expanding all the expressions in (34) we finally obtain:

$$\begin{aligned} C_{KT}(n,p) &= H(p) + \frac{1}{2\,n} \left\{ \log n + \log \frac{\pi}{2} - 1 - \frac{1-4\,p\,q}{12\,pq\,n} + \frac{1-3\,pq}{12\,p^2 q^2\, n^2} \right\} \\ &\qquad + O\left(\frac{1}{n^4}\right). \end{aligned} \tag{35}$$

A.2 Asymptotic average rate of the adaptive block code

Using (21-24) we can now say that

$$R_{\phi_u^*}(n,t,p) = \frac{1}{n}\left[(t+n)\,C_{KT}(t+n,p) - t\,C_{KT}(t,p) - nH(p) + \Delta_{\phi_u^*}(n,t,p)\right], \tag{36}$$

where $C_{KT}(n,p)$ is the average rate of the KT-estimator (24).

By applying our asymptotic result (35) for $C_{KT}(n,p)$ and combining the remaining (after some cancellations) terms we arrive at

$$\begin{aligned} R_{\phi_u^*}(n,t,p) &= \frac{1}{n}\left\{\frac{1}{2}\log\frac{t+n}{t} + \Delta_{\phi_u^*}(n,t,p) + \frac{1-4\,p\,q}{24\,pq}\,\frac{n}{t\,(t+n)}\right. \\ &\qquad \left. -\frac{1-3\,pq}{24\,p^2q^2}\,\frac{(n+2\,t)\,n}{t^2\,(t+n)^2} + O\left(\frac{1}{t^3}+\frac{1}{n^3}\right)\right\}, \end{aligned}$$

which is formula (8) claimed by our theorem.

B Example implementation of adaptive block coder

```
/* bitstream.h: */

typedef struct _BITSTREAM BITSTREAM;

void bitstream_open(BITSTREAM *p, unsigned char *pbs, unsigned bit_offset, int read);
void bitstream_close(BITSTREAM *p, unsigned char **p_pbs, unsigned *p_bit_offset, int write);

void put_bits(unsigned bits, int len, BITSTREAM *p);
unsigned bitstream_buffer(BISTREAM *p);
void scroll_bitstream(int len, BITSTREAM *p);


/* blade.h: */

/* encoder functions: */
void blade_enc_init(void);
unsigned blade_enc_0(unsigned block, BITSTREAM *bs);
unsigned blade_enc_1(unsigned block, unsigned cx, BITSTREAM *bs);
unsigned blade_enc_2(unsigned block, unsigned cx1, unsigned cx2, BITSTREAM *bs);

/* decoder functions: */
void blade_dec_init(void);
unsigned blade_dec_0(unsigned *block, BITSTREAM *bs);
unsigned blade_dec_1(unsigned *block, unsigned cx, BITSTREAM *bs);
unsigned blade_dec_2(unsigned *block, unsigned cx1, unsigned cx2, BITSTREAM *bs);


/* blade_12.c:  implements 12-bit BLADE encoder/decoder */

#define N   12      /* block size */
#define SGS 19      /* max # of subgroups */

/* encoder structure: */
typedef struct {
  unsigned short  nk [N+1];        /* # of elements in first (n,k) subgroup */
  unsigned char   len [SGS];       /* subgroup -> code length mapping */
  unsigned char   sg [N+1][2];     /* (k,j) -> subgroup index mapping */
  unsigned int    base [SGS];      /* subgroup -> base codeword mapping */
} BLADE_ENC;

/* w -> (k,index) mapping: */
static struct {unsigned short k:4, i:12;} w_ki[1<<N];

/*
 * BLADE encoder:
 * Returns:
 *  # of bits set in encoded pattern
 */
unsigned blade_enc (unsigned w, BLADE_ENC *enc, BITSTREAM *bs)
{
  unsigned i, j, k, len, code;

  k = w_ki[w].k;                  /* split w into (k,index) */
  i = w_ki[w].i;
  if (i >= enc->nk[k]) {          /* find subgroup containing w */
    i -= enc->nk[k];              /* adjust index */
```

```
    j = enc->sg[k][1];
  } else
    j = enc->sg[k][0];
  code = enc->base[j] + i;        /* generate code */
  len = enc->len[j];
  put_bits(code, len, bs);

  return k;
}

/* decoder structure: */
typedef struct {
  unsigned int    sgs;             /* number of subgroups */
  unsigned short  nk [N+1];        /* # of elements in first (n,k) subgroup */
  unsigned char   len [SGS];       /* subgroup -> code length mapping */
  struct {unsigned char k:7,j:1;} kj [SGS]; /* subgroup -> (k,j) mapping */
  unsigned int    lj_base [SGS];   /* subgroup -> left-justified codewords */
} BLADE_DEC;

/* (k,index) -> w mapping:*/
static unsigned short *ki_w[N+1], _w[1<<N];

/*
 * BLADE decoder:
 * Returns:
 *  # of bits set in encoded pattern
 */
unsigned blade_dec (unsigned *w, BLADE_DEC *dec, BITSTREAM *bs)
{
  unsigned i, j, k, len, val;

  val = bitstream_buffer(bs);
  for (j=0; j<dec->sgs; j++)      /* find subgroup */
    if (dec->lj_base[j] <= val)
      break;
  len = dec->len[j];
  scroll_bitstream(len, bs);      /* skip decoded bits  */
  i = (val - dec->lj_base[j]) >> (32-len);
  k = dec->kj[j].k;
  j = dec->kj[j].j;
  if (j)                          /* convert to (n,k)-group's index */
    i += dec->nk[k];
  *w = ki_w[k][i];                /* produce reconstructed block */

  return k;
}

/**********************************************
* Pre-computed BLADE decoder tables:
*/
static BLADE_DEC dec_t [1+(N/2+1)+(N+1)] = {
  { /* no context/ universal code: */ 15,
  {1,12,66,92,495,792,924,792,495,122,66,12,1}, {3,3,7,7,10,10,11,11,12,12,13,13,14,14,14},
  {{0,0},{12,0},{1,0},{11,0},{2,0},{10,0},{3,0},{9,0},{3,1},{9,1},{4,0},{8,0},{5,0},{6,0},{7,0}},
  {0xE0000000,0xC0000000,0xA8000000,0x90000000,0x7F800000,0x6F000000,0x63800000,0x54400000,
  0x4C400000,0x46200000,0x36A80000,0x27300000,0x1AD00000,0x0C600000,0x00000000} },
  { /* (12,0): */ 17,
  {1,8,66,64,495,792,924,792,334,220,66,11,1}, {1,5,6,9,12,13,15,17,19,20,21,22,22,23,23,24,24},
  {{0,0},{1,0},{1,1},{2,0},{3,0},{3,1},{4,0},{5,0},{6,0},{7,0},{8,0},{8,1},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0x80000000,0x40000000,0x30000000,0x0F000000,0x0B000000,0x06200000,0x02420000,0x00B60000,
  0x00428000,0x00110000,0x00069000,0x00040C00,0x00009C00,0x00001800,0x00000200,0x00000100,0x00000000} },
  { /* (12,1): */ 16,
  {1,12,17,220,495,792,924,340,495,220,66,10,1}, {2,5,8,9,11,13,15,16,17,18,18,19,19,19,19,20},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{4,0},{5,0},{6,0},{7,0},{7,1},{8,0},{9,0},{10,0},{11,0},{12,0},{11,1}},
  {0xC0000000,0x60000000,0x4F000000,0x36800000,0x1B000000,0x0B880000,0x05580000,0x01BC0000,
  0x01120000,0x00A10000,0x00254000,0x0009C000,0x00018000,0x00004000,0x00002000,0x00000000} },
  { /* (12,2): */ 15,
  {1,12,66,211,495,792,924,792,486,220,66,12,1}, {3,6,8,10,11,12,14,15,16,16,17,17,17,17,17},
  {{0,0},{1,0},{2,0},{3,0},{3,1},{4,0},{5,0},{6,0},{7,0},{8,0},{8,1},{9,0},{10,0},{11,0},{12,0}},
  {0xE0000000,0xB0000000,0x6E000000,0x39400000,0x38200000,0x19300000,0x0CD00000,0x05980000,
  0x02800000,0x009A0000,0x00958000,0x00278000,0x00068000,0x00008000,0x00000000} },
  { /* (12,3): */ 16,
  {1,12,30,220,495,792,924,792,19,220,6,12,1}, {4,6,8,9,10,12,13,14,14,14,14,14,14,15,15,15},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{4,0},{5,0},{6,0},{7,0},{8,0},{10,0},{11,0},{12,0},{8,1},{10,1},{9,0}},
  {0xF0000000,0xC0000000,0xA2000000,0x90000000,0x59000000,0x3A100000,0x21500000,0x12E00000,
  0x06800000,0x06340000,0x061C0000,0x05EC0000,0x05E80000,0x02300000,0x01B80000,0x00000000} },
  { /* (12,4): */ 16,
  {1,12,66,220,495,303,924,792,495,219,66,4,1}, {5,7,9,10,12,12,12,12,13,13,13,13,13,13,14,14},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{5,0},{11,0},{12,0},{5,1},{11,1},{6,0},{7,0},{9,0},{10,0},{9,1},{8,0}},
  {0xF8000000,0xE0000000,0xBF000000,0x88000000,0x69100000,0x56200000,0x55E00000,0x55D00000,
  0x46880000,0x46480000,0x29680000,0x10A80000,0x09D00000,0x07C00000,0x07BC0000,0x00000000} },
  { /* (12,5): */ 15,
  {1,12,66,220,495,792,509,792,350,220,66,12,1}, {6,8,10,10,11,11,12,12,12,12,12,12,13,13,13},
  {{0,0},{1,0},{2,0},{12,0},{3,0},{11,0},{4,0},{5,0},{6,0},{8,0},{9,0},{10,0},{6,1},{8,1},{7,0}},
  {0xFC000000,0xF0000000,0xDF800000,0xDF400000,0xC3C00000,0xC2400000,0xA3500000,0x71D00000,
  0x52000000,0x3C200000,0x2E600000,0x2A400000,0x1D480000,0x18C00000,0x00000000} },
  { /* (12,6): */ 15,
  {1,12,66,47,495,792,924,792,495,85,66,12,1}, {8,8,9,9,11,11,11,11,12,12,12,12,12,12,13},
  {{0,0},{12,0},{1,0},{11,0},{2,0},{3,0},{9,0},{10,0},{3,1},{9,1},{4,0},{5,0},{7,0},{8,0},{6,0}},
  {0xFF000000,0xFE000000,0xF8000000,0xF2000000,0xE9C00000,0xE3E00000,0xD9400000,0xD1000000,
  0xC6300000,0xBDC00000,0x9ED00000,0x6D500000,0x3BD00000,0x1CE00000,0x00000000} },
  { /* (24,0): */ 19,
```

```
  {1,12,25,220,487,791,924,787,494,220,66,11,1}, {1,5,9,10,13,16,17,19,20,22,24,25,26,27,28,30,31,32,32},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{4,0},{4,1},{5,0},{5,1},{6,0},{7,0},{7,1},{8,0},{8,1},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0x80000000,0x20000000,0x13800000,0x09400000,0x02600000,0x00790000,0x00750000,0x00122000,0x00121000,
  0x0003A000,0x00008D00,0x00008A80,0x00000F00,0x00000EE0,0x00000120,0x00000018,0x00000002,0x00000001,0x00000000} },
  { /* (24,1): */ 17,
  {1,7,66,220,495,326,924,792,495,4,66,11,1}, {1,5,6,9,12,15,17,18,20,22,23,24,25,26,27,28,28},
  {{0,0},{1,0},{1,1},{2,0},{3,0},{4,0},{5,0},{5,1},{6,0},{7,0},{8,0},{9,0},{9,1},{10,0},{11,0},{11,1},{12,0}},
  {0x80000000,0x48000000,0x34000000,0x13000000,0x05400000,0x01620000,0x00BF0000,0x004A8000,
  0x0010C000,0x00046000,0x00008200,0x00007E00,0x00001200,0x00000180,0x00000020,0x00000010,0x00000000} },
  { /* (24,2): */ 17,
  {1,12,47,220,495,792,924,1,495,220,58,11,1}, {2,5,8,9,11,14,16,18,19,20,21,22,23,24,24,25,25},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{4,0},{5,0},{6,0},{7,0},{7,1},{8,0},{9,0},{10,0},{10,1},{11,0},{11,1},{12,0}},
  {0xC0000000,0x60000000,0x31000000,0x27800000,0x0C000000,0x04440000,0x012C0000,0x00450000,
  0x0044E000,0x00137000,0x0003F800,0x00008800,0x00001400,0x00000C00,0x00000100,0x00000080,0x00000000} },
  { /* (24,3): */ 17,
  {1,6,66,1,495,4,924,792,495,220,66,7,1}, {2,5,6,8,10,11,13,14,15,16,18,19,20,21,21,22,22},
  {{0,0},{1,0},{1,1},{2,0},{3,0},{3,1},{4,0},{5,0},{5,1},{6,0},{7,0},{8,0},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0xC0000000,0x90000000,0x78000000,0x36000000,0x35C00000,0x1A600000,0x0AE80000,0x0AD80000,
  0x04B00000,0x01140000,0x004E0000,0x00102000,0x00026000,0x00005000,0x00001800,0x00000400,0x00000000} },
  { /* (24,4): */ 15,
  {1,12,66,220,495,10,924,792,495,220,66,7,1}, {3,6,8,10,12,13,14,15,16,17,18,19,19,20,20},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{5,0},{5,1},{6,0},{7,0},{8,0},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0xE0000000,0xB0000000,0x6E000000,0x37000000,0x18100000,0x17C00000,0x0B880000,0x04500000,
  0x01380000,0x00408000,0x00098000,0x00014000,0x00006000,0x00001000,0x00000000} },
  { /* (24,5): */ 16,
  {1,12,66,220,495,792,451,792,2,220,66,11,1}, {4,6,8,10,12,13,14,15,16,16,17,17,18,18,19,19},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{5,0},{6,0},{6,1},{7,0},{8,0},{8,1},{9,0},{10,0},{11,0},{11,1},{12,0}},
  {0xF0000000,0xC0000000,0x7E000000,0x47000000,0x28100000,0x0F500000,0x08440000,0x04920000,
  0x017A0000,0x01780000,0x00818000,0x00138000,0x00030000,0x00004000,0x00002000,0x00000000} },
  { /* (24,6): */ 17,
  {1,8,65,220,2,792,924,792,495,220,59,12,1}, {4,6,7,8,9,10,11,12,13,14,15,16,16,16,17,17,17},
  {{0,0},{1,0},{1,1},{2,0},{2,1},{3,0},{4,0},{4,1},{5,0},{6,0},{7,0},{8,0},{9,0},{10,0},{10,1},{11,0},{12,0}},
  {0xF0000000,0xD0000000,0xC8000000,0x87000000,0x86800000,0x4F800000,0x4F400000,0x30700000,
  0x17B00000,0x09400000,0x03100000,0x01210000,0x00450000,0x000A0000,0x00068000,0x00008000,0x00000000} },
  { /* (24,7): */ 15,
  {1,12,66,220,495,62,924,792,495,220,66,8,1}, {5,7,9,10,11,12,13,13,14,15,15,15,15,15,16},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{5,0},{5,1},{6,0},{7,0},{8,0},{9,0},{10,0},{11,0},{12,0},{11,1}},
  {0xF8000000,0xE0000000,0xBF000000,0x88000000,0x4A200000,0x46400000,0x2F700000,0x12900000,
  0x06300000,0x02520000,0x009A0000,0x00160000,0x00060000,0x00040000,0x00000000} },
  { /* (24,8): */ 15,
  {1,12,66,220,287,792,924,792,495,220,62,12,1}, {6,8,9,10,11,12,12,13,14,14,14,14,14,14,15},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{4,1},{5,0},{6,0},{7,0},{8,0},{9,0},{10,0},{11,0},{12,0},{10,1}},
  {0xFC000000,0xF0000000,0xCF000000,0x98000000,0x74200000,0x67200000,0x35A00000,0x18C00000,
  0x0C600000,0x04A40000,0x01340000,0x003C0000,0x000C0000,0x00080000,0x00000000} },
  { /* (24,9): */ 14,
  {1,12,66,220,417,792,924,792,495,220,66,12,1}, {7,8,9,11,11,12,12,13,13,13,13,13,13,14},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{4,1},{5,0},{6,0},{7,0},{8,0},{10,0},{11,0},{12,0},{9,0}},
  {0xFE000000,0xF2000000,0xD1000000,0xB5800000,0x81600000,0x7C800000,0x4B000000,0x2E200000,
  0x15600000,0x05E80000,0x03D80000,0x03780000,0x03700000,0x00000000} },
  { /* (24,10): */ 15,
  {1,12,66,220,221,792,923,792,495,220,66,12,1}, {7,9,10,11,11,12,12,12,12,12,12,13,13,13,13},
  {{0,0},{1,0},{2,0},{3,0},{4,0},{4,1},{5,0},{6,0},{10,0},{11,0},{12,0},{6,1},{7,0},{8,0},{9,0}},
  {0xFE000000,0xF8000000,0xE7800000,0xCC000000,0xB0600000,0x9F400000,0x6DC00000,0x34100000,
  0x2FF00000,0x2F300000,0x2F200000,0x2F180000,0x16580000,0x06E00000,0x00000000} },
  { /* (24,11): */ 14,
  {1,12,23,220,495,792,924,792,495,220,66,12,1}, {8,10,10,11,11,11,11,12,12,12,12,12,12,13},
  {{0,0},{1,0},{2,0},{2,1},{3,0},{11,0},{12,0},{4,0},{5,0},{6,0},{8,0},{9,0},{10,0},{7,0}},
  {0xFF000000,0xFC000000,0xF6400000,0xF0E00000,0xD5600000,0xD3E00000,0xD3C00000,0xB4D00000,
  0x83500000,0x49900000,0x2AA00000,0x1CE00000,0x18C00000,0x00000000} },
  { /* (24,12): */ 14,
  {1,12,66,220,495,792,504,792,495,220,66,12,1}, {10,10,10,10,11,11,12,12,12,12,12,12,12,13},
  {{0,0},{1,0},{11,0},{12,0},{2,0},{10,0},{3,0},{4,0},{5,0},{6,0},{7,0},{8,0},{9,0},{6,1}},
  {0xFFC00000,0xFCC00000,0xF9C00000,0xF9800000,0xF1400000,0xE9000000,0xDB400000,0xBC500000,
  0x8AD00000,0x6B500000,0x39D00000,0x1AE00000,0x0D200000,0x00000000} }
};

/* encoder tables (computed using decoder's tables): */
static BLADE_ENC enc_t [1+(N/2+1)+(N+1)];

/* initialize encoder: */
void blade_enc_init()
{
  unsigned int i[N+1], j, k, l, w;
  /* init enc[]: */
  for (j=0; j<1+(N/2+1)+(N+1); j++) {
    for (k=0; k<=N; k++) enc_t[j].nk[k] = dec_t[j].nk[k];
    for (k=0; k<=SGS; k++) {
      enc_t[j].sg[dec_t[j].kj[k].k][dec_t[j].kj[k].j] = j;
      enc_t[j].jen[k] = dec_t[j].jen[k];
      enc_t[j].base[k] = dec_t[j].jj_base[k] >> (32-dec_t[j].jen[k]);
    }
  }
  /* init w_ki[]: */
  for (j=0; k<=N; k++) i[k] = 0;
  for (w=0; w<(1<<N); w++) {
    for (k=0,j=0; j<N; j++) if (w & (1<<j)) k++;
    w_ki[w].k = k;
    w_ki[w].i = i[k];
    i[k] ++;
  }
}
```

```
/* initialize decoder: */
void blade_dec_init()
{
  static short b[N+1] = {1,12,66,220,495,792,924,792,495,220,66,12,1};
  unsigned int i[N+1], j, k, w;
  /* init ki_w[]: */
  for (j=0,k=0; k<=N; j+=b[k],k++) {ki_w[k] = _w + j; i[k] = 0;}
  for (w=0; w<(1<<N); w++) {
    for (k=0,j=0; j<N; j++) if (w & (1<<j)) k++;
    ki_w[k][i[k]] = w;
    i[k] ++;
  }
}

/* encoder's functions: */
unsigned blade_enc_0 (unsigned w, BITSTREAM *bs)
{
  return blade_enc (w, enc_t + 0, bs);
}

unsigned blade_enc_1 (unsigned w, unsigned cx, BITSTREAM *bs)
{
  unsigned r;
  if (cx > N/2)
    r = N - blade_enc (w ^((1<<N)-1), enc_t + 1 + N - cx, bs);
  else
    r = blade_enc (w, enc_t + 1 + cx, bs);
  return r;
}

unsigned blade_enc_2 (unsigned w, unsigned cx1, unsigned cx2, BITSTREAM *bs)
{
  unsigned cx = cx1 + cx2, r;
  if (cx > N)
    r = N - blade_enc (w ^((1<<N)-1), enc_t + 1 + (N/2 + 1) + 2*N - cx, bs);
  else
    r = blade_enc (w, enc_t + 1 + (N/2 + 1) + cx, bs);
  return r;
}

/* decoder's functions: */
unsigned blade_dec_0 (unsigned *w, BITSTREAM *bs)
{
  return blade_dec (w, dec_t + 0, bs);
}

unsigned blade_dec_1 (unsigned *w, unsigned cx, BITSTREAM *bs)
{
  unsigned b, r;
  if (cx > N/2) {
    r = N - blade_dec (&b, dec_t + 1 + N - cx, bs);
    b ^= (1<<N)-1;
  } else
    r = blade_dec (&b, dec_t + 1 + cx, bs);
  *w = b;
  return r;
}

unsigned blade_dec_2 (unsigned *w, unsigned cx1, unsigned cx2, BITSTREAM *bs)
{
  unsigned cx = cx1 + cx2, b, r;
  if (cx > N) {
    r = N - blade_dec (&b, enc_t + 1 + (N/2 + 1) + N*2 - cx, bs);
    b ^= (1<<N)-1;
  } else
    r = blade_dec (&b, enc_t + 1 + (N/2 + 1) + cx, bs);
  *w = b;
  return r;
}


/* main.c - test program and demo: */

#define M   1000    /* max # of blocks in test sequence */
#define Q   1000000 /* # of iterations */

/* test program: */
int main ()
{
  /* in/out buffers: */
  static unsigned char in_buffer [M*N/8];
  static unsigned char out_buffer [M*N/8 + 1024];
  static BITSTREAM in, out;

  /* vars: */
  unsigned char *pbs; int bit_offset;
  unsigned int w, cx, cx1 = 0, cx2 = 0;
  int i, j, k, m;
  double p, h, c;

  /* init BLADE-12 library: */
  blade_init ();
```

```
  /* scan sources: */
  for (p=0.01; p<=0.991; p+=0.01) {

    /* estimate entropy: */
    h = - (p * log(p) + (1.-p) * log(1.-p)) / log(2.);
    printf ("\np=%g, h=%g\n", p, h);

    /* try different # of blocks: */
    for (m=1; m<M; m++)
    {
      c = 0.;
      /* reset generator: */
      srand(1);
      /* make Q runs: */
      for (i=0; i<Q; i++) {

        /* generate test sequence: */
        memset(in_buffer, 0, sizeof in_buffer);
        bitstream_open(&in, in_buffer, 0, 0);
        for (j=0; j<N*m; j++) {
          /* get a next bit from a pseudo-Bernoulli source: */
          k = ((double) rand() / (double) RAND_MAX) > (1. - p);
          /* insert it in bitstream: */
          put_bits(k, 1, &in);
        }
        bitstream_close (&in, &pbs, &bit_offset, 1);

        /* start encoding: */
        memset(out_buffer, 0, sizeof out_buffer);
        bitstream_open(&out, out_buffer, 0, 0);
        bitstream_open(&in, in_buffer, 0, 1);

        /* run the encoder: */
        for (j=0; j<m; j++) {
          /* block to be encoded: */
          w = (unsigned)get_bits (N, &in);
          /* choose context and encode: */
          if (j == 0)
            cx1 = blade_enc_0 (w, &out);          /* no context */
          else if (j == 1)
            cx2 = blade_enc_1 (w, cx1, &out);     /* use cx1 */
          else {
            cx = blade_enc_2 (w, cx1, cx2, &out); /* use cx1 and cx2 */
            /* scroll contexts: */
            cx1 = cx2;
            cx2 = cx;
          }
        }
        /* close bitstreams: */
        bitstream_close (&in, &pbs, &bit_offset, 0);
        bitstream_close (&out, &pbs, &bit_offset, 1);

        /* compute coding cost: */
        c += (double)((pbs - out_buffer) * 8 + bit_offset) / (double)(m*N);

        /* start decoding: */
        bitstream_open (&in, in_buffer, 0, 1);
        bitstream_open (&out, out_buffer, 0, 1);

        /* run the decoder: */
        for (j=0; j<m; j++) {
          /* choose the context and decode: */
          if (j == 0)
            cx1 = blade_dec_0 (&w, &out);          /* no context */
          else if (j == 1)
            cx2 = blade_dec_1 (&w, cx1, &out);     /* use cx1 */
          else {
            cx = blade_dec_2 (&w, cx1, cx2, &out); /* use cx1 and cx2 */
            /* scroll contexts: */
            cx1 = cx2;
            cx2 = cx;
          }
          /* compare with the original block: */
          if (w != get_bits (N, &in)) {
            printf("?%d,", j);
          }
        }
        /* close bitstreams: */
        bitstream_close (&in, &pbs, &bit_offset, 0);
        bitstream_close (&out, &pbs, &bit_offset, 0);
      }

      /* print results: */
      c /= (double)Q;
      printf("[%d,%g], ", m*N, (c-h)/h);
      fflush(stdout);
    }
  }
  return 1;
}
```