A Fast and Efficient Nearly-Optimal Adaptive Fano Coding Scheme

Article *in* Information Sciences · June 2006

DOI: 10.1016/j.ins.2005.07.010 · Source: DBLP

2 authors, including:

Luis Rueda
University of Windsor
240 PUBLICATIONS **2,188** CITATIONS

SEE PROFILE

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Sciences 176 (2006) 1656–1683
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

A fast and efficient nearly-optimal adaptive Fano coding scheme

Luis Rueda [a], B. John Oommen [b,*]

[a] *School of Computer Science, University of Windsor, 401 Sunset Avenue, Windsor, ON, Canada N9B 3P4*
[b] *School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, ON, Canada K1S 5B6*

Received 8 February 2005; received in revised form 14 June 2005; accepted 13 July 2005

Abstract

Adaptive coding techniques have been increasingly used in lossless data compression. They are suitable for a wide range of applications, in which *on-line* compression is required, including communications, internet, e-mail, and e-commerce. In this paper, we present an adaptive Fano coding method applicable to binary and multi-symbol code alphabets. We introduce the corresponding partitioning procedure that deals with consecutive partitionings, and that possesses, what we have called, the *nearly-equal-probability property*, i.e. that satisfy the principles of Fano coding. To determine the optimal partitioning, we propose a brute-force algorithm that searches the entire space of *all* possible partitionings. We show that this algorithm operates in polynomial-time complexity on the size of the input alphabet, where the degree of the polynomial is given by the size of the output alphabet. As opposed to this, we also propose a greedy algorithm that quickly finds a sub-optimal, but accurate, consecutive partitioning. The empirical results on real-life benchmark data files demonstrate that our scheme compresses and decompresses faster than adaptive Huffman coding, while consuming less memory resources.

* Corresponding author. Tel.: +1 613 520 2600x4358; fax: +1 613 520 4334.
E-mail addresses: lrueda@cs.uwindsor.ca (L. Rueda), oommen@scs.carleton.ca (B.J. Oommen).

0020-0255/$ - see front matter
doi:10.1016/j.ins.2005.07.010

Keywords: Adaptive coding; Fano coding; Data compression

1. Introduction

Data encoding involves processing an input sequence, $\mathscr{X} = x[1]\cdots x[M]$, where each input symbol, $x[i]$, is drawn from a source alphabet, $\mathscr{S} = \{s_i, \ldots, s_m\}$ whose probabilities are $\mathscr{P} = [p_1, \ldots, p_m]$ with $2 \leqslant m < \infty$. The encoding process is rendered by transforming $\mathscr{X}$ into an output sequence, $\mathscr{Y} = y[1]\cdots y[R]$, where each output symbol, $y[i]$, is drawn from a code alphabet, $\mathscr{A} = \{a_1, \ldots, a_r\}$. The main problem in lossless data compression is to find an encoding scheme that minimizes the size of $\mathscr{Y}$, in such a way that $\mathscr{X}$ is completely recovered by the decompression process.

Encoding methods can be implemented either statically or adaptively. The most well-known static coding techniques are Huffman's algorithm [1], Fano's method [2], and arithmetic coding [3,4]. The static Huffman coding can be implemented as a recursive algorithm that proceeds by generating the so-called Huffman tree [3]. It recursively merges symbols (nodes) into a new conceptual symbol which constitutes an internal node of the tree. In this way, Huffman's algorithm generates a "coding" tree in a bottom-up fashion. After the tree is generated, the code words can be obtained by labeling the branches of the tree using the symbols from the code alphabet in such a way that the encoder and decoder use the same labeling scheme.

The static Fano's method proceeds by generating a coding tree as well, but in a top-down fashion. At each step, the list of symbols is partitioned into two (or more, if the output alphabet is non-binary) new sublists, generating two or more new nodes in the corresponding coding tree. Although Fano's method, typically, generates a sub-optimal encoding scheme, the loss in compression with respect to Huffman coding is minimal [2]. As in Huffman coding, the code words can be obtained in a similar fashion, i.e. by arbitrarily assigning code alphabet symbols to the branches of the tree.

On the other hand, adaptive coding is important since the data is encoded by performing a *single* pass assuming that $\mathscr{P} = [p_1, \ldots, p_m]$ are unknown, as opposed to the static algorithms which require two passes—the first to *learn* the probabilities, and the second to *accomplish* the encoding. Conversely, adaptive coding is the best choice in many applications that require on-line compression such as in communication networks, LANs, internet applications, e-mail, ftp, and e-commerce.

Adaptive coding techniques include Huffman coding [1], which was independently proposed by Faller in 1973 [5] and Gallager in 1978 [6], and augmented later by Knuth [7] and Vitter [8], and *arithmetic coding* [3,4]. Other adaptive coding methods include *interval coding* and *recency rank encoding* [3,9].

Adaptive coding approaches that use higher-order statistical models, and other structural models, include *dictionary techniques* (LZ and its enhancements) [10–12], *prediction with partial matching* (PPM) and its enhancements [13,14], *block sorting compression* [15,16], and *grammar based compression* (GBC) [17].

One of the major drawbacks of the adaptive Huffman coding scheme is that it requires the maintenance of an on-line Huffman *tree* which has to be updated each time a symbol is encoded. These memory requirements do not seem to be a serious problem for state-of-the-art computers, where a few kilobytes is insignificant. The real problem arises when we are dealing with Huffman coding involving higher-order modeling, since a kth-order model requires the maintenance of the statistical information about the different contexts. Thus, when k is large, the number of contexts grows, and so does the size of the data structure required for encoding/decoding.

As opposed to Huffman coding, Fano's method requires only the probabilities (or frequency counters) for the *contexts* that have already appeared, and only these need to be maintained at any given time. This is even more advantageous in other structural/statistical models as well, such as the LZW algorithm [18,19], which uses large tables to maintain the dictionary of matched phrases, PPM, utilizing a considerable amount of space (typically, in the order of magnitude of dozens of megabytes) to store the statistical information of the source symbols for the different contexts [20,21]. Viewed from this perspective, the use of any adaptive approach for Fano coding is quite advantageous since it typically requires a fraction of the memory required to maintain the corresponding Huffman tree, even though their space complexities are the same. This is the focus of this paper.

Although Fano's method, typically, generates a sub-optimal encoding scheme,[1] the loss[2] in the compression ratio with respect to Huffman's algorithm is relatively small. However, as pointed out in [23], if the method is modified to also consider list rearrangement strategies, nearly-optimal compression can be achieved.

In this paper, we present a *greedy adaptive version* of Fano's method for binary and multi-symbol code alphabets, which apart from being important in its own right from an academic point of view, is specially suitable for applications in which memory constraints are tight. This paper details the encoding and decoding algorithms which invoke the partitioning procedure. We introduce the corresponding greedy partitioning procedure that deals with consecutive

[1] This is due to the fact that determining the optimal partitioning in Fano's method is NP-hard [22].

[2] Note that this "loss" refers to how much a system would have compressed, if using Huffman coding. However, in both Huffman and Fano coding the compression is *lossless*, i.e. the original data is *completely* recovered from the compressed data.

partitioning possessing, what we have called, the *nearly-equal-probability property*. In this case, unlike the binary case, since the number of partitions is combinatorially large, the greedy algorithm is linear, but sub-optimal. The paper also contains formal proofs of our claims. The application of these techniques to real-life image data is currently under investigation.

2. Adaptive Fano coding

As discussed earlier, Fano's method is a well-known method which has been studied in its static form only [2,3]. Indeed, to the best of our knowledge, the adaptive version of this method has not been proposed yet. Since adaptive coding is quite important in many applications, we propose the adaptive version of Fano coding. We propose algorithms for encoding and decoding, and for partitioning for binary and multi-symbol code alphabets. In this section, we present the encoding and decoding procedures, and in subsequent sections, we discuss two different approaches for partitioning, one that follows a ''brute-force'' approach, and the other that uses a ''greedy'' procedure that leads to a sub-optimal scheme.

2.1. A brute-force method for adaptive Fano coding

An adaptive approach for the Fano coding could be done by a scheme analogous to the original static Fano coding algorithm at each encoding step. For each symbol that appears in the input, we can apply the Fano coding method, and thus, obtain the code word for each source symbol. The code word that corresponds to the input symbol is then sent to the output, and the probabilities or frequency counters for the input symbols are updated so that they are used for the next encoding step. The decoder is designed in such a way that it resembles the encoding algorithm by generating all the code words (or a Fano tree) using Fano's method, and choosing the symbol for which the input bits match the corresponding code word. This *brute-force* approach can be shown to losslessly compress and decompress sequences drawn from universal sources, since Fano's method possesses the property of generating *prefix* codes.

There is no doubt that this method, although correct, is really of a brute-force nature. First of all, it requires the resolution of the entire code word set at every encoding/decoding step. In that sense, it resembles a search problem which visits the entire search space to compute a solution. In this case, this event of ''visiting'' the search space is indeed, one of creating the entire codeword set. What we intend to achieve in this paper is to devise a scheme that avoids the reconstruction of the *entire* code word set at each encoding/decoding step. To do this, we propose a greedy approach that does not require the maintenance of a coding/decoding *tree*. We simultaneously require that the

proposed scheme generates *only* the code word that corresponds to the symbol being encoded/decoded.

2.2. The greedy encoding algorithm

Consider the source alphabet $\mathscr{S} = \{s_1, \ldots, s_m\}$, where $2 \leqslant m < \infty$, with probabilities of occurrence $\mathscr{P} = [p_1, \ldots, p_m]$, the code alphabet $\mathscr{A} = \{a_1, \ldots, a_r\}$, and the input sequence $\mathscr{X} = x[1] \cdots x[M]$. First of all, we assume that we start with arbitrary (unknown) probabilities for the source symbols. In order to hasten the encoding process, and for ease of explanation, the estimates of the probabilities of occurrence are considered as *frequency counters*,[3] and initialized to unity.

The implementation of the encoding procedure for the greedy adaptive version for Fano coding is shown in Algorithm **Greedy_Adaptive_Fano_Encoding**. At each time step, a list that contains the frequency counters for the source alphabet symbols, $\mathscr{P}(k) = \{p_1(k), \ldots, p_m(k)\}$, is maintained. The starting list, $\mathscr{P}(1)$, is initialized as if all the source symbols are equally likely. Whenever $x[k]$ is encoded, the partitioning procedure is invoked, which returns the corresponding output code alphabet symbol. The design of an efficient partitioning procedure for $r > 2$ is a fairly involved task, which will be discussed later, in detail, in Sections 3 and 4. Note that in the implementation, the list of symbols, $\mathscr{S}(k)$, can be omitted. This is due to the fact that each symbol can be mapped to an integer in the range $[0, \ldots, m]$. We also note that the partitioning procedures for encoding and decoding are slightly different. However, in order to make the presentation simpler, we merely present a single partitioning algorithm in which we highlight the statements that should be enabled/disabled in either encoding or decoding.

Algorithm 1. Greedy_Adaptive_Fano_Encoding

Input: The source alphabet, $\mathscr{S}$. The input sequence, $\mathscr{X}$.
Output: The output sequence, $\mathscr{Y}$.
Method:
$\mathscr{S}(1) \leftarrow \mathscr{S}$; Allocate m cells for $\mathscr{P}(1)$
for $i \leftarrow$ **1 to** m **do** // *Initialize the frequency counters*
$p_i(1) \leftarrow 1$
endfor
$j \leftarrow 1$
for $k \leftarrow 1$ **to** M **do** // *For every symbol of the source sequence*
$t \leftarrow 1$; $b \leftarrow m$; $p(k) \leftarrow \sum_{i=1}^{m} p_i(k)$

[3] We alternatively use the term "probabilities of occurrence" to also refer to these frequencies.

while $b > t$ **do**
$y[j] \leftarrow \text{Partition}(\mathscr{S}(k), \mathscr{P}(k), \mathscr{A}, x[k], t, b, p(k))$
$j \leftarrow j + 1$
endwhile
Let i be the index of $s_i(k) = x[k]$.
Increment $p_i(k)$.
endfor
end Algorithm Greedy_Adaptive_Fano_Encoding

2.3. The greedy decoding algorithm

The decoding procedure is similar to the encoder. Instead of probabilities, we again maintain a frequency counter for each symbol, which records the number times that this symbol has appeared so far in the input sequence. The decoding procedure is detailed in Algorithm **Greedy_Adaptive_Fano_Decoding**. A partitioning procedure, which is similar to that invoked by the encoder, is now utilized to render the two processes synchronized.

Note that, as discussed earlier, no extra information other than the encoded sequence is provided to the encoder. This is, in fact, an important advantage over the static methods which require that some extra information (such as the symbol probabilities, the decoding tree, etc.) be stored in the overhead of the encoded sequence. The analogous statement is true for the decoder.

Algorithm 2. Greedy_Adaptive_Fano_Decoding

Input: The encoded sequence, $\mathscr{Y}$. The source alphabet, $\mathscr{S}$.
Output: The source sequence, $\mathscr{X}$.
Method:
$\mathscr{S}(1) \leftarrow \mathscr{S}$; Allocate m cells for $\mathscr{P}(1)$
for $i \leftarrow 1$ **to** m **do**
$p_i(1) \leftarrow 1$
endfor
$k \leftarrow 1$; $t \leftarrow 1$; $b \leftarrow m$; $p(1) \leftarrow \sum_{i=1}^{m} p_i(1)$
for $j \leftarrow 1$ **to** R **do**
$\text{Partition}(\mathscr{S}(k), \mathscr{P}(k), \mathscr{A}, y[j], t, b, p(k))$
if $t = b$ **then**
$x[k] \leftarrow s_t(k)$
Increment $p_t(k)$
$k \leftarrow k + 1$; $t \leftarrow 1$; $b \leftarrow m$; $p(k) \leftarrow \sum_{i=1}^{m} p_i(k)$
endif
endfor
end Algorithm Greedy_Adaptive_Fano_Decoding

We now observe some important properties which are crucial when implementing the encoding and decoding algorithms. First, the encoder and the decoder must use the same labelling scheme. Second, they both have to start with the same initial probabilities. Third, they must use the same probability updating procedure. Finally, the encoder uses $\mathscr{P}(k)$ to encode $x[k]$, which, obviously, is known by the decoder too.

To speed up the compression and decompression phases, in the actual implementation, we maintain a list that contains *only* the symbols that have occurred so far. To achieve this, when a new symbol occurs, a flag, which corresponds to the code for a *not yet transmitted* (NYT) symbol, is sent to the output, followed by the ASCII code word corresponding to the new symbol. When the decoder encounters the flag for an NYT symbol, it just reads the next eight bits, and reconstructs the new symbol. We do not claim to pioneer this approach; it has previously been utilized in adaptive Huffman coding [24].

2.4. Tree-based adaptive Fano coding

Apart from the above-described method, it is conceivable that a tree-based adaptive version of Fano coding can also be devised. This could be done by maintaining a Fano coding tree, which is used to generate the code word for the current input symbol. After the probabilities (or rather, the estimated frequencies) of the input symbols change, the Fano tree may also change, and hence it has to be updated. The decoder must thus maintain the Fano tree for decoding, replicate the probability updating procedure, and also follow tree updating rules carried out by the encoder. Although this method seems to be more efficient than our greedy approach, we prefer to rather use a *list* with the symbols and their associated probabilities, for various reasons. First, the Fano tree needs additional memory, and we propose to demonstrate that our algorithm is applicable to scenarios in which memory requirements are tight. Second, unlike Huffman's coding, the static Fano coding does not necessarily generate a tree, because the code words can be obtained without generating a tree. Third, using a list provides a much more flexible implementation that allows using different updating probability procedures without changing the encoding and decoding rules. To be more specific, the adaptive Huffman coding approaches proposed by Knuth and Vitter [7,8] work only when frequency counters are maintained, and at each encoding step, the frequency of the symbol that has occurred is increased by unity. Other probability updating approaches might require specific tree updating procedures. While this is a limitation for the latter, our adaptive Fano coding approach can be implemented with *any* probability updating procedure that uses a list of symbols and their probabilities. In this regard, we are currently investigating the use of *stochastic learning* techniques in adaptive data compression.

3. Nearly-equal-probability partitioning

It is well known that binary codes are of practical importance, mainly because they are the ones used by most digital devices. However, r-ary codes, such as ternary codes [25], have been studied extensively, because of their cryptographic and error-correcting capabilities. In this paper, we focus mainly on the adaptive Fano coding that uses r-ary code alphabets, and discuss the binary code alphabet afterwards, as a particular case. Designing a partitioning procedure for a code alphabet $\mathscr{A} = \{a_1, \ldots, a_r\}$, where $r > 2$, is far from trivial. First of all, we observe that Huffman's algorithm requires that the number of source alphabet symbols is $m = r + \kappa(r-1)$ for some positive integer κ. Fano's method also requires that this constraint be satisfied. However, this constraint should be satisfied not only for the source alphabet, but also for *all the sublists* obtained after partitioning.

To clarify issues, we consider the source alphabet $\mathscr{S} = \{s_1, \ldots, s_m\}$, the probabilities of occurrence $\mathscr{P} = [p_1, \ldots, p_m]$, and the code alphabet $\mathscr{A} = \{a_1, \ldots, a_r\}$. We also consider Algorithm **Greedy_Adaptive_Fano_Encoding** which requires that $\mathscr{S}$ and $\mathscr{P}$ be partitioned into r sublists $\{\mathscr{S}_1, \ldots, \mathscr{S}_r\}$, such that $|\mathscr{S}_j| = \kappa(r-1) + 1$ for some positive integer κ. The output symbol, $y[j]$, is to be a_i where $x[k] \in \mathscr{S}_i$. This partitioning scheme is also required by Algorithm **Greedy_Adaptive_Fano_Decoding**. The partitioning procedure that takes care of these constraints is designed below.

In order to design a suitable partitioning procedure, we assume that the components of each partitioning are consecutive in the original partitioned list. We have opted to derive consecutive partitionings in order to reduce the required amount of computation. Non-consecutive partitionings constitute a more general case that can even (slightly) improve compression. However, for an alphabet of size m, 2^m possible partitionings (or subsets) have to be generated. The definition of consecutive partitioning is formally given below.

Definition 1. Consider the source alphabet $\mathscr{S} = \{s_1, \ldots, s_m\}$, where $m = r + \kappa(r-1)$ for some integer $\kappa \geqslant 0$, and the code alphabet $\mathscr{A} = \{a_1, \ldots, a_r\}$. Suppose that $\mathscr{S}$ is partitioned into $\{\mathscr{S}_1, \mathscr{S}_2, \ldots, \mathscr{S}_r\}$. The partitioning is said to be *consecutive* if and only if $\mathscr{S}_1 = \{s_1, \ldots, s_{i_1}\}, \mathscr{S}_2 = \{s_{i_1+1}, \ldots, s_{i_2}\}, \ldots, \mathscr{S}_r = \{s_{i_{r-1}+1}, \ldots, s_m\}$, where $1 \leqslant i_1 < i_2 < \cdots < i_{r-1} < m$, and $|\mathscr{S}_j| = \kappa_j(r-1) + 1$ for integers $\kappa_j \geqslant 0$, $j = 1, \ldots, r$.

When considering consecutive partitionings, there are $\frac{(\frac{m-r}{r-1}+r-1)!}{(\frac{m-r}{r-1})!(r-1)!}$ possible partitionings, where m is the size of the source alphabet and r is the size of the code alphabet. This is stated in the following theorem, whose proof is given in Appendix B.

Theorem 1. *Let $\mathscr{S} = \{s_1, \ldots, s_m\}$ be the source alphabet and $\mathscr{A} = \{a_1, \ldots, a_r\}$ be the code alphabet, where $m = r + \kappa(r-1)$ for some integer $\kappa \geqslant 0$, and $r \geqslant 2$. The number of possible consecutive partitionings is given by*

$$\binom{\frac{m-r}{r-1} + r - 1}{r-1} = \frac{\left(\frac{m-r}{r-1} + r - 1\right)!}{\left(\frac{m-r}{r-1}\right)!(r-1)!}. \tag{1}$$

After determining the number of all possible consecutive partitionings, we are only interested in the ones that satisfy the principles of Fano coding, i.e. we would like to ensure that the sums of the probabilities of the sublists resulting from the partitioning are *as nearly equal* as possible. As in the binary-alphabet case, we define this property for a multi-symbol code alphabet, $\mathscr{A} = \{a_1, \ldots, a_r\}$, where $r \geqslant 2$. This property is formally stated in the following definition.

Definition 2. Let $\{\mathscr{S}_1, \mathscr{S}_2, \ldots, \mathscr{S}_r\}$ be a set obtained by a consecutive partitioning on $\mathscr{S}$. Let q_j be the sum of probabilities of $\mathscr{S}_j$, and q be the sum of probabilities of $\mathscr{S}$. $\{\mathscr{S}_1, \mathscr{S}_2, \ldots, \mathscr{S}_r\}$ has the *nearly-equal-probability property*, if and only if

$$d(\{\mathscr{S}_1, \mathscr{S}_2, \ldots, \mathscr{S}_r\}) = \sum_{j=1}^{r} \left| q_j - \frac{q}{r} \right|^{\alpha} \tag{2}$$

is minimal, where α is a real number such that $\alpha \geqslant 1$, and $d(\cdot)$ denotes the *distance function*.

We now introduce an example that clarify these issues.

Example 1. Consider the source alphabet $\mathscr{S} = \{a, b, c, d, e, f, g\}$ whose probabilities of occurrence are $\mathscr{P} = [.28, .22, .2, .1, .1, .05, .05]$, and the code alphabet $\mathscr{A} = \{0, 1, 2\}$.

Using (1), we see that there are $\binom{\frac{7-3}{3-1} + 3 - 1}{3-1} = 6$ possible consecutive partitionings. The six partitionings, the sums of their corresponding probabilities, and the distance as calculated from (2) for $\alpha = 1$, are listed in the following table:

$\{\mathscr{S}_1, \mathscr{S}_2, \mathscr{S}_3\}$	$q_j = \sum_{s_i \in \mathscr{S}_j} p_i, j = 1, 2, 3$	$d(\{\mathscr{S}_1, \mathscr{S}_2, \mathscr{S}_3\})$
{{a}, {b}, {c,d,e,f,g}}	**.28**, **.22**, **.5**	**0.33**
{{a}, {b,c,d}, {e,f,g}}	.28, .52, .2	0.37
{{a}, {b,c,d,e,f}, {g}}	.28, .67, .05	0.67
{{a,b,c}, {d}, {e,f,g}}	.7, .1, .2	0.73
{{a,b,c}, {d,e,f}, {g}}	.7, .25, .05	0.73
{{a,b,c,d,e}, {f}, {g}}	.9, .05, .05	1.13

Observe that there is one optimal consecutive partitioning (i.e. the one that satisfies the nearly-equal-probability property stated in Definition 2), and is the one found in the first row of the table, namely, $\{\{a\},\{b\},\{c,d,e,f,g\}\}$, whose distance is $\sum_{j=1}^{3}|q_j-\frac{1}{3}|\simeq 0.33$. The optimal algorithm would attempt to compute this partitioning, and this can be achieved by a brute-force polynomial-time algorithm presented momentarily. We shall later show that by a linear searching scheme, a sub-optimal, but accurate, partitioning can be obtained.

We now introduce a "brute-force" algorithm that searches the space of *all* possible consecutive partitionings, and computes those that satisfy the nearly-equal-probability property, i.e. the ones whose distance is minimal. The formal procedure is shown below in Algorithm **All_Consecutive_Partitionings**. Let P_i be the cumulative probability of $\{s_1,\ldots,s_i\}$, which is calculated as $P_i=\sum_{j=1}^{i}p_j$. Using these cumulative probabilities, we avoid recalculating $q_j=\sum_{s_i\in\mathscr{S}_j}p_i$ at each step of the algorithm. The way by which this algorithm generates all possible consecutive partitionings of $\mathscr{S}$ is based on the proof of Theorem 1. It proceeds by enumerating all the $r-1$ valid partitioning indices, $i_1,\ldots,i_{r-1}$. This is achieved by the recursive procedure Permute( ), which maintains a list of all valid partitioning indices, and for each permutation, it computes the corresponding distance as in (2). Note that the list L, initially empty, contains $r-1$ elements which are indexed from 0 to $r-2$. The partitioning procedure is written in such a way that it can be invoked by both encoding and decoding modules. Thus, the statements marked with $*^1$, $*^2$ and $*^4$ must be disabled if invoked from the decoder, and the one marked with $*^3$ must be enabled so that the indices are identified using the code alphabet symbol $y[j]$.

This algorithm enumerates all possible consecutive partitionings. This is stated in the theorem given below, whose proof is given in Appendix B.

Algorithm 3. All_Consecutive_Partitionings

Input: The source alphabet, $\mathscr{S}$, and probabilities, $\mathscr{P}$. The code alphabet, $\mathscr{A}$. The sum of probabilities of $\mathscr{P}$, q.
Output: The new range, $[t, b]$. The output symbol, $y[j]$ (for the encoder only). The sum of probabilities of the new $\mathscr{P}$, p.
Method:

```
procedure Partition(S, P, A: list; s: symbol; var t, b: integer; var p: real);
    m_p ← b − t + 1
    P_{t−1} ← 0; κ ← (m_p − r)/(r − 1); d_min ← MAX_VALUE
    for i ← t to b do
        P_i ← P_{i−1} + p_i
    endfor
```

```
    {i_1, ..., i_{r-1}}_min ← Permute([ ], P, 0, κ, r, p)
    {i_1, ..., i_{r-1}}_min+ ← {t} ∪ {i_1, ..., i_{r-1}}_min ∪ {b}
    *1  i ← index of s in 𝒮 // Disable for decoding
    *2  [t, b] ← [i_w, i_{w+1}], such that i_w ⩾ i and i_{w+1} ⩽ i // Disable for
    decoding
    *3  // Enable for decoding: j ← index of s in 𝒜; t ← i_j; b ← i_{j+1}
    p ← P_b − P_{t−1}
    *4  return a_w // Disable for decoding
  endprocedure

  procedure Permute(L, P: list; start, κ, r: integer; p: real)
    last ← size of L
    Append start to L
    for i ← start to κ do
      L[last] ← i
      if last < r − 2 then
        Permute(L, i, κ, r)
      else
        d ← 0; i_j ← 0
        for j ← 1 to r do
          i_{j−1} ← i_j
          if j < r then
            i_j ← L[j] * (r − 1) + j
          else
            i_j ← m
          endif
            d ← d + |P_{i_j} − P_{i_{j−1}} − p/r|
        endfor
        if d < d_min then
          d_min ← d; {i_1, ..., i_{r−1}}_min ← {i_1, ..., i_{r−1}}
        endif
      endif
    endfor
    return {i_1, ..., i_{r−1}}_min
  endprocedure
end Algorithm All_Consecutive_Partitionings
```

Theorem 2. *Let* $\mathscr{S} = \{s_1, \ldots, s_m\}$ *be the source alphabet whose probabilities of occurrence are* $\mathscr{P} = [p_1, \ldots, p_m]$, *and let* $\mathscr{A} = \{a_1, \ldots, a_r\}$ *be the code alphabet, where* $m = r + \kappa(r-1)$ *for some integer* $\kappa \geqslant 0$. *Algorithm* **All_Consecutive_Partitionings** *enumerates all possible consecutive partitionings.*

Although Algorithm **All_Consecutive_Partitionings** enumerates and searches the entire space of all possible consecutive partitionings, its worst-case time complexity is bounded by a polynomial whose degree is given by the number of code alphabet symbols. This fact is stated and proved in the following theorem, whose proof is given in Appendix B.

Theorem 3. *Let $\mathscr{S} = \{s_1, \ldots, s_m\}$ be the source alphabet whose probabilities of occurrence are $\mathscr{P} = [p_1, \ldots, p_m]$, and let $\mathscr{A} = \{a_1, \ldots, a_r\}$ be the code alphabet, where $m = r + \kappa(r-1)$ for some integer $\kappa \geqslant 0$. The worst-case time complexity of Algorithm* **All_Consecutive_Partitionings** *is*

(*i*) $O(m^{r-1})$ *if* $r < m$, *and*
(*ii*) $O(m)$ *if* $r = m$.

In order to implement the greedy adaptive Fano coding for multi-symbol code alphabets, the optimal consecutive partitioning has to be performed in each sublist in which the source symbol being encoded, $x[k]$, is contained. The number of partitioning steps for a single symbol is given by the length of the code word sent to the output. In the worst case, this number can take values up to $m-1$. Hence the worst-case time complexity of encoding a source symbol is $O(m^r)$. Therefore, this procedure is quite inefficient for large code alphabets (even for $r = 3$ or 4). To overcome this situation, we propose a greedy but accurate, linear-time algorithm that finds a sub-optimal partitioning.

4. A greedy sub-optimal nearly-equal-probability partitioning

4.1. The multi-symbol case

Since the brute-force algorithm that we presented in Section 3 is quite inefficient, we now present a non-exhaustive, greedy algorithm that finds a *sub-optimal* consecutive partitioning[4]. The implementation of this procedure is depicted in Algorithm **Greedy_*r*-ary_Adaptive_Fano_Partitioning**. Using this partitioning procedure, a source symbol can be encoded in $O(m)$ time. The source alphabet, the probabilities of occurrence and the code alphabet are as usual, and $x[k]$ is the symbol being encoded at time k. The current range, $[t, b]$, contains the boundaries of $\mathscr{S}$ (or eventually the sublist) in which $x[k]$ is contained. After the partitioning procedure is performed, these boundaries represent the sub-sublist in which $x[k]$ is contained. The variable p stands for

[4] Although, it is quite non-traditional to submit such a procedure that is common for both the encoder and the decoder, where certain statements are inhibited for one type of invocation, we have found this quite useful in the main application of our scheme: *on-line coding*.

the sum of probabilities of those symbols between t and b. After performing the partitioning procedure, p contains the sum of probabilities of the resulting sub-sublist.

Algorithm **Greedy_r-ary_Adaptive_Fano_Partitioning** proceeds by partitioning the list into two sublists. The variable p_{temp} keeps track of the cumulative probability. The while loop is executed until p_{temp} is greater than or equal to $\frac{p}{r}$. If these two sublists do not satisfy the nearly-equal-probability property, the first $r-1$ elements of the second list have to be moved to the first sublist. In fact, if $\frac{p}{r} - p_{\text{temp}} + \frac{p_{\text{temp1}}}{2} < 0$, p_{temp1} (the last $r-1$ components added to p_{temp}) have to be moved to the first sublist.

Algorithm 4. Greedy_r-ary_Adaptive_Fano_Partitioning

Input: The source alphabet, $\mathscr{S}$, and probabilities, $\mathscr{P}$. The code alphabet, $\mathscr{A}$.
The current source or code alphabet symbol, s. The current range, $[t, b]$.
The sum of probabilities of $\mathscr{P}$, p.

Output: The new range, $[t, b]$. The output symbol, $y[j]$ (for the encoder only).
The sum of probabilities of the new $\mathscr{P}$, p.

Method:

```
procedure Partition(𝒮, 𝒫, 𝒜: list; s: symbol; var t, b: integer; var p: real);
  i ← t; found ← false; p_temp ← 0; r ← |𝒜|
  while not (found) do
    *1  if s_i = s then found ← true endif // Disable for decoding
    p_temp ← p_temp + p_i; i ← i + 1
    while p_temp < p/r do
      p_temp1 ← 0
      for j ← 1 to r − 1 do
        p_temp1 ← p_temp1 + p_i; i ← i + 1
        *2  if s_i = s then found ← true endif // Disable for decoding
      endfor
      p_temp ← p_temp + p_temp1
    endwhile
    *3  // Enable for decoding: if a_{|𝒜|−r+1} = s then found ← true endif
    if p/r − p_temp + p_temp1/2 < 0 then
    i ← i − r + 1; p_temp ← p_temp − p_temp1
    endif
    if found then
    b ← i − 1; p ← p_temp
    else  t ← i; p ← p − p_temp
    if r = 1 then
    found ← true
    else      r ← r − 1; p_temp ← 0   endif
```

```
        endif
      endwhile
      *4  return a_{|𝒜|−r+1} // Disable for decoding
    endprocedure
end Algorithm Greedy_r-ary_Adaptive_Fano_Partitioning
```

If the symbol being encoded is found in the first sublist, the iteration stops, and a code alphabet symbol is returned. If it is not found, unity is subtracted from r, and the second sublist is partitioned in the same fashion. The labeling strategy used here consists of assigning a_1 to the first sublist, a_2 to the second sublist, and so on. The partitioning procedure given in Algorithm **Greedy_r-ary_Adaptive_Fano_Partitioning** is designed to be invoked by the encoding algorithm. The corresponding procedure for the decoding process uses similar rules, without returning any code alphabet symbol. As in the brute-force partitioning procedure, the algorithm is also written as a general procedure applicable to both the encoding and decoding processes. Thus, the statements marked with $*^1$, $*^2$ and $*^4$ must be disabled if invoked from the decoder, and the one marked with $*^3$ must be enabled so that the search stops when the encoded symbol s is found.

To conclude this section, we note that the theoretical framework that we propose in this paper can be extended in numerous ways. For example, the time complexity of our partitioning scheme can be substantially improved by using the so-called *Fenwick's data structure* [4], which is used to efficiently maintain and access the cumulative frequency counters. In terms of space requirements, this implies the use of a *single word* per symbol to store the statistical information about the source. However, our empirical results for the binary-alphabet case show the superiority of our scheme when compared to adaptive Huffman coding.

4.2. The binary case

4.2.1. The algorithms for the binary case

Since the binary alphabet is a special case of the multi-symbol alphabet, it is clear that the partitioning procedure discussed in the previous section applies also to the binary code alphabet. However, since some of the "loops" disappear in this special case, and the conditions become trivial, we believe that it can be advantageous to include the algorithm for the binary case for the following reasons:

- The partitioning algorithm for binary alphabets is much simpler and easier to understand.

- These algorithms are the ones that have been implemented to yield the empirical results shown later in the paper. Including them here permits other researchers to validate our results.[5]

The encoding and decoding algorithms are included in Appendix A.

4.2.2. Complexity analysis for the binary case

We shall first analyze the time complexity for the encoding algorithm. The first **for** loop is executed m times. Inside the second **for** loop, which is executed M times, there is a **while** loop. This loop contains the invocation of the partitioning procedure which takes $|\mathscr{S}_0|$ steps. Hence, if $x[k]$ is in $\mathscr{S}_0$, the total number of steps to encode $x[k]$ is not greater than m, because in the worst case, it would take up to $|\mathscr{S}_0|$ more steps to find $x[k]$. If $x[k]$ is not in $\mathscr{S}_0$, it is searched in $\mathscr{S}_1$, which takes up to m steps (in the worst case, when $x[k]$ is the last element of $\mathscr{S}$). Therefore, given the source alphabet $\mathscr{S} = \{s_1, \ldots, s_m\}$ and the input sequence $\mathscr{X} = x[1] \cdots x[M]$, in the worst case, the number of steps that Algorithm **Greedy_Adaptive_Fano_Encoding** takes to generate the output is $O(mM)$ steps.

The time complexity of Algorithm **Greedy_Adaptive_Fano_Decoding** is derived as in the encoding algorithm. The first **for** loop, which initializes the probabilities, takes m steps. The main **for** loop, which takes R steps, invokes the partitioning procedure, where R is the number of bits in the output sequence, $\mathscr{Y}$. Since the partitioning procedure invoked by the decoder resembles that of the encoder, to decode each source symbol, $x[k]$, the partitioning takes M steps, where M is the number of symbols in the original sequence. Therefore, given the source alphabet $\mathscr{S} = \{s_1, \ldots, s_m\}$, Algorithm **Greedy_Adaptive_Fano_Decoding** decodes an encoded sequence $\mathscr{Y} = y[1] \cdots y[R]$ in $O(mM)$ steps.

The space complexity of Algorithms **Greedy_Adaptive_Fano_Encoding** and **Greedy_Adaptive_Fano_Decoding** is relatively small, since they require the maintenance of *only* a list of symbols, and their associated probabilities or frequencies. Each symbol and probability require constant space. Thus, the space complexity of these algorithms is $O(m)$.

4.2.3. Implementation considerations

From the theoretical analysis given above, we observe that both the adaptive Huffman coding and the greedy adaptive Fano coding require $O(m)$ space. But the maintenance and overhead for the former is significantly more expensive than for the Fano data structure—since the latter is maintained essentially as a list. Thus, while the Huffman tree requires $O(m)$ space, it still needs to maintain $2m - 1$ nodes. Each node, contains a symbol, a frequency counter,

[5] The source code can be made available upon request.

a pointer to its parent, pointers to its left and right children, and pointers to the next and previous nodes in the list. This is significantly more than what is required by the corresponding data structure used in the greedy adaptive Fano coding. For example, consider the case in which the source alphabet is the ASCII alphabet. The tree contains up to 511 nodes. Typically, each node requires the following amount of space: one byte for the symbol, four bytes for the frequency counter, and two bytes for each pointer.[6] Thus, the adaptive Huffman coding data structure requires 7665 bytes, as opposed to the 1280 bytes required by the greedy adaptive Fano coding data structure, where each element in the array contains the symbol and its frequency counter. This implies that the adaptive Fano coding requires about *one-sixth* of the memory used by the adaptive Huffman coding. This advantage increases significantly if higher-order models or dictionary-based structures are used.

5. Correctness of the greedy adaptive Fano Coding

In this section, we present the analysis for the correctness of the greedy adaptive Fano coding for multi-symbol code alphabets using the sub-optimal, linear-time partitioning procedure presented in Section 4. We shall show that given any source sequence, the output generated by encoding using the partitioning procedure given in Algorithm **Greedy_*r*-ary_Adaptive_Fano_Partitioning** has enough information to reproduce the original sequence by performing the reverse process, decoding, using the same partitioning procedure. We also show that this model for multi-symbol code alphabets achieves on-line compression. The two theorems, whose proofs are in Appendix B, are given below.

Theorem 4. *Consider the source alphabet* $\mathscr{S} = \{s_1, \ldots, s_m\}$, *and the code alphabet* $\mathscr{A} = \{a_1, \ldots, a_r\}$. *Suppose that the input sequence* $\mathscr{X} = x[1] \cdots x[M]$ *is encoded by invoking the partitioning procedure of Algorithm* **Greedy_*r*-ary_Adaptive_Fano_Partitioning** *yielding an output sequence* $\mathscr{Y} = y[1] \cdots y[R]$. *If* $\mathscr{Y}$ *is decoded by invoking the same partitioning procedure into* $\mathscr{X}^{\star} = x^{\star}[1] \cdots x^{\star}[T]$, *then* $\mathscr{X}^{\star} = \mathscr{X}$ (*i.e.* $T = M$, *and* $x^{\star}[i] = x[i]$, $\forall i$, $i = 1, \ldots, M$.

Theorem 5. *Under the conditions of Theorem* 4, *Algorithm* **Greedy_*r*-ary_Adaptive_Fano_Partitioning** *achieves on-line compression and decompression.*

[6] We assume that the tree is stored in an array, and hence two bytes are required for each pointer. However, if the pointer field contain a reference to a memory address, each pointer would require four bytes.

Though the contribution of this paper is essentially theoretical, in the next section, we show that it has significant applications. Other practical implications of this in an image processing application (where the alphabet is typically of size 256) are currently being studied.

6. Empirical results

In order to test the compression power and the compression/decompression speed of our schemes, we have implemented the binary-alphabet version of the greedy adaptive Fano coding, as given in Appendix A, using the C programming language. The source code for the encoder and the decoder have been compiled using Forte C, version 6.2, and then tested on a Sun machine, model sun4u, which runs a SunOS 5.8 operating system. To compare the greedy adaptive version of Fano's method with the adaptive Huffman coding, we ran both prototypes on files of the well-known standard benchmarks, the Calgary corpus[7] and the Canterbury corpus [26]. To be fair in the comparison, we have used the *compact* utility from Unix,[8] which is an implementation of the adaptive Huffman coding as in [6]. Note that we do not compare these methods with any other higher-order model, such as LZ, PPM or block-sorting, since the latter constitute statistical/structural models which *must* use a "back-end" source coding method such as Huffman or Fano coding. We thus compare the Huffman and Fano coding methods using a *zeroth-order* model.

The empirical results obtained from these runs are cataloged in Tables 1 and 2 respectively. The labels 'AH' and 'AF' refer to the results of the Unix compact utility and the greedy adaptive Fano coding approach respectively. The name and the original size of the files are given in the first and second columns respectively. The columns labelled 'Comp. (%)' correspond to the percentage of compression, calculated as $(1 - \frac{\text{compressed\_size}}{\text{original\_size}}) \cdot 100$ for AH and AF respectively. The columns labelled 'Encoding time' and 'Decoding time' contain the time (in seconds) used to compress the original file and decompress the compressed file respectively, while the column labelled 'Gain' represents the time taken by AH divided by that of AF.

Observe that the greedy adaptive Fano's method compresses marginally less efficiently than the adaptive Huffman coding. Huffman coding compressed just 0.14% more than Fano's method on the files of the Calgary corpus. This difference is slightly less, 0.13%, on the files of the Canterbury corpus. This difference in favor of Huffman coding is present in all the files except in "paper1", in which Fano's method compressed 0.02% more than Huffman coding. This

[7] Electronically available at ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/.

[8] The highest hurdle we encountered in compiling these results was that of testing the algorithms objectively against standard benchmark (as opposed to our own) *implementations*.

Table 1
An empirical comparison of the greedy adaptive version of Fano's method and the adaptive Huffman coding which were tested on files of the Calgary corpus

File name	Original size	Comp. (%)		Encoding time			Decoding time		
		AH	AF	AH	AF	Gain	AH	AF	Gain
bib	111,261	34.46	34.37	0.2674	0.1472	1.82	0.2134	0.1365	1.56
book1	768,771	42.96	42.87	1.3082	0.7171	1.82	0.9939	0.6434	1.54
book2	610,856	39.67	39.51	1.1019	0.6328	1.74	0.8373	0.5715	1.47
geo	102,400	28.80	28.52	0.2841	0.2020	1.41	0.2330	0.1906	1.22
news	377,109	34.60	34.49	0.7518	0.4426	1.70	0.5810	0.4063	1.43
obj1	21,504	24.19	23.16	0.1132	0.0604	1.87	0.1014	0.0556	1.82
obj2	246,814	21.19	21.10	0.6228	0.4711	1.32	0.4853	0.4372	1.11
paper1	53,161	37.00	37.02	0.1567	0.0772	2.03	0.1326	0.0672	1.97
progc	39,611	34.19	33.89	0.1367	0.0665	2.06	0.1159	0.0618	1.88
progl	71,646	39.82	39.38	0.1863	0.0923	2.02	0.1543	0.0841	1.83
progp	49,379	38.51	38.07	0.1482	0.0743	1.99	0.1262	0.0688	1.83
trans	93,695	30.20	30.16	0.2440	0.1390	1.76	0.1980	0.1249	1.59
Total/avg.	2,546,207	36.82	36.68	5.3213	3.1225	1.70	4.1723	2.8479	1.47

Table 2
Empirical results obtained after running the adaptive Huffman coding and the greedy adaptive Fano's method on files of the Canterbury corpus

File name	Original size	Comp. (%)		Encoding time			Decoding time		
		AH	AF	AH	AF	Gain	AH	AF	Gain
alice29.txt	148,481	42.96	42.73	0.3045	0.1529	1.99	0.2439	0.1393	1.75
asyoulik.txt	125,179	39.32	39.20	0.2752	0.1496	1.84	0.2244	0.1349	1.66
cp.html	24,603	33.61	33.60	0.1064	0.0496	2.15	0.0959	0.0469	2.04
fields.c	11,550	35.87	35.52	0.0812	0.0336	2.42	0.0959	0.0469	2.04
grammar.lsp	3721	39.56	38.73	0.0637	0.0238	2.68	0.0758	0.0318	2.38
kennedy.xls	1,029,744	55.03	54.80	1.5049	1.0479	1.44	1.1670	0.9866	1.18
lcet10.txt	419,235	41.78	41.77	0.7546	0.4195	1.80	0.5831	0.3819	1.53
plrabn12.txt	471,162	43.47	43.36	0.8205	0.4432	1.85	0.6348	0.4021	1.58
ptt5	513,216	79.21	79.18	0.4348	0.1741	2.50	0.3537	0.1740	2.03
xargs.1	4277	36.24	36.12	0.0635	0.0250	2.54	0.0658	0.0237	2.78
Total/avg.	2,751,168	53.85	53.72	4.4093	2.5192	1.75	3.5403	2.3681	1.49

behavior, which seems illogical since Huffman coding achieves optimal compression if the probabilities were exact, is because both methods use different probability updating procedures.

Although their worst-case time complexities are the same, when it concerns real-life compression times, we observe that in all cases, Fano's method is faster than Huffman coding—the latter requires 70% and 75% more time than the

former to compress all the files of the Calgary corpus and the Canterbury corpus respectively. This is reasonable since the adaptive Fano coding does not need to navigate and maintain a tree. In the case of the decompression phase, Fano's method is faster than Huffman coding, where the latter takes almost 50% more time than the former to decompress all the files. The gain in the decompression phase is smaller due to the fact that Huffman coding decodes in a top down fashion, avoiding the double pass required in the encoding process.

7. Conclusions

In this paper, we have presented a "greedy" adaptive version of Fano's method for multi-symbol code alphabets. We have developed the encoding, decoding and partitioning procedures for the multi-symbol source alphabet and the multi-symbol code alphabet.

We then studied the problem of obtaining consecutive partitionings that satisfy the principles of Fano coding. To find the optimal partitioning, we have proposed a brute-force algorithm that searches the entire space of *all* possible partitionings. We have shown that this algorithm operates in the worst case, in $O(m^{r-1})$ time. As opposed to this, we also proposed a greedy algorithm that yields a sub-optimal but accurate consecutive partitioning. The corresponding results for the correctness, and the on-line compression achieved by using this procedure have been stated and formally proven.

The empirical results on files of the standard benchmarks show that our approach compresses marginally less than adaptive Huffman coding, but that is markedly faster in both compression and decompression. We are currently investigating the application of these techniques in image processing applications.

The problem of devising a non-brute-force algorithm that finds the *optimal* consecutive partitioning remains open. Another open problem that deserves investigation is that of devising a theoretical upper bound on the length of the code achieved by our greedy partitioning scheme and its relationship to the *ideal* partitioning scheme.

Acknowledgements

They authors would like to thank Prof. Daniel Hirschberg from UCI for providing a fast, standard executable version of the compact utility for Unix. They would also like to thank the anonymous referees for their constructive comments. This research work has been partially supported by NSERC, the Natural Sciences and Engineering Research Council of Canada, CFI,

the Canadian Foundation for Innovation, and OIT, the Ontario Innovation Trust.

Appendix A. Encoding and decoding algorithms: Binary case

Algorithm 5. Greedy_Adaptive_Fano_Binary_Encoding

Input: The source alphabet and the initial probabilities, $\mathscr{S}$ and $\mathscr{P}$ respectively. The input sequence, $\mathscr{X}$.
Output: The output sequence, $\mathscr{Y}$.
Method:

procedure Partition($\mathscr{S}, \mathscr{P}, \mathscr{A}$: list; s: symbol; **var** t, b: integer; **var** p: real);
  $i \leftarrow t$; *found* $\leftarrow$ false; $p_{\text{temp}} \leftarrow 0$
  while $p_{\text{temp}} < \frac{p}{2}$ **do**
    if $s_i = s$ **then** *found* $\leftarrow$ true **endif**
    $p_{\text{temp}} \leftarrow p_{\text{temp}} + p_i$; $i \leftarrow i + 1$
  endwhile
  if $\frac{p}{2} - p_{\text{temp}} + \frac{p_{i-1}}{2} < 0$ **then**
    $i \leftarrow i - 1$; $p_{\text{temp}} \leftarrow p_{\text{temp}} - p_i$
  endif
  if *found* **then**
    $b \leftarrow i - 1$; $p \leftarrow p_{\text{temp}}$; **return** 0
  else
    $t \leftarrow i$; $p \leftarrow p - p_{\text{temp}}$; **return** 1
  endif
endprocedure
$\mathscr{S}(1) \leftarrow \mathscr{S}$; $\mathscr{P}(1) \leftarrow \mathscr{P}$
for $i \leftarrow 1$ **to** m **do** // *Initialize the frequency counters*
  $p_i \leftarrow 1$
endfor
$j \leftarrow 1$
for $k \leftarrow 1$ **to** M **do** // *For every symbol of the source sequence*
  $t \leftarrow 1$; $b \leftarrow m$; $p \leftarrow \sum_{i=1}^{m} p_i$
  while $b > t$ **do**
    $y[j] \leftarrow \text{Partition}(\mathscr{S}(k), \mathscr{P}(k), \mathscr{A}, x[k], t, b, p)$
    $j \leftarrow j + 1$
  endwhile
  Increment the weight of $x[k]$.
  Swap $x[k]$ and the top-most symbol in $\mathscr{S}(k)$ whose weight is less than that of $x[k]$. (Do the same updating for $\mathscr{P}(k)$)
endfor
end Algorithm Greedy_Adaptive_Fano_Binary_Encoding

Algorithm 6. Greedy_Adaptive_Fano_Binary_Decoding

Input: The encoded sequence, $\mathscr{Y}$. The source alphabet and the initial probabilities, $\mathscr{S}$ and $\mathscr{P}$ respectively.
Output: The source sequence, $\mathscr{X}$.
Method:

```
procedure Partition(S, P, A: list; a: symbol; var t, b: integer; var p: real);
    p_temp ← 0; i ← t
    while p_temp < p/2 do
        p_temp ← p_temp + p_i; i ← i + 1
    endwhile
    if p/2 − p_temp + p_{i−1}/2 < 0 then
        i ← i − 1; p_temp ← p_temp − p_i
    endif
    if a = 0 then
        b ← i − 1; p ← p_temp
    else
        t ← i; p ← p − p_temp
    endif
endprocedure
S(1) ← S; P(1) ← P
for i ← 1 to m do
    p_i ← 1
endfor
k ← 1; t ← 1; b ← m; p ← Σ_{i=1}^{m} p_i
for j ← 1 to R do
    Partition(S(k), P(k), A, y[j], t, b, p)
    if t = b then
        x[k] ← s_t(k)
        Increment the weight of x[k]
        Swap x[k] and the top-most symbol in S(k) whose counter is less
        than that of x[k]. (Do the same updating for P(k))
        k ← k + 1; t ← 1; b ← m; p ← Σ_{i=1}^{m} p_i
    endif
endfor
end Algorithm Greedy_Adaptive_Fano_Binary_Decoding
```

Appendix B. Proofs

Proof of Theorem 1. Using the definition of consecutive partitionings (Definition 1), $\mathscr{S}$ is partitioned into

$$\mathscr{S}_1 = \{s_1, \ldots, s_{i_1}\}, \mathscr{S}_2 = \{s_{i_1+1}, \ldots, s_{i_2}\}, \ldots, \mathscr{S}_r = \{s_{i_{r-1}+1}, \ldots, s_m\},$$

where $1 \leqslant i_1 < i_2 < \cdots < i_{r-1} < i_m$, and $|\mathscr{S}_j| = \kappa_j(r-1)+1$, for integers $\kappa_j \geqslant 0$, $j = 1, \ldots, r$.

We now observe the following:

(i) Since $\mathscr{S}_1$ has $\kappa_1(r-1)+1$ symbols for some integer $\kappa_1 \geqslant 0$,and s_1 and s_{i_1} are the first and the last elements of $\mathscr{S}_1$ respectively, it implies that $i_1 = \kappa_1(r-1)+1$, where $\kappa_1 = 0, \ldots, \kappa$.
(ii) Again, $\mathscr{S}_2$ has also $\kappa_2(r-1)+1$ symbols, for some integer $\kappa_2 \geqslant 0$. Thus s_{i_1+1} and s_{i_2} are the first and the last elements of $\mathscr{S}_2$. This implies that $i_2 = \kappa_2(r-1)+2$, where $\kappa_2 = 0, \ldots, \kappa$, and $\kappa_2 \geqslant \kappa_1$.

Continuing in the same fashion, we have

$(r-1)$ Since $\mathscr{S}_{r-1}$ has also $\kappa_{r-1}(r-1)+1$ symbols, for some integer $\kappa_{r-1} \geqslant 0$, and $s_{i_{r-2}+1}$ and s_{i_r-1} are the first and last elements of $\mathscr{S}_{r-1}$ respectively, we have $i_{r-1} = \kappa_{r-1}(r-1)+(r-1)$, where $\kappa_{r-1} = 0, \ldots, \kappa$, and $\kappa_{r-1} \geqslant \kappa_{r-2}$.

Let i_j be a *valid partitioning index*. Also, let the jth sublist resulting from the partitioning be $\mathscr{S}_j = \{s_{i_{j-1}+1}, \ldots, s_{i_j}\}$.

From the arguments above, we observe that

$$\kappa_j = \kappa_j(r-1)+j, \quad \text{where } \kappa_j = 0, \ldots, \kappa, \text{ and } j = 1, \ldots, r-1.$$

We also know that $1 \leqslant i_1 < i_2 < \cdots < i_{r-1} < m$, which implies that

$$1 \leqslant \kappa_1(r-1)+1 < \kappa_2(r-1)+2 < \cdots < \kappa_{r-1}(r-1)+(r-1) < m. \tag{3}$$

Subtracting unity from all the terms of the inequality, we have

$$0 \leqslant \kappa_1(r-1) < \kappa_2(r-1)+1 < \cdots < \kappa_{r-1}(r-1)+(r-2) < m-1. \tag{4}$$

Since all the quantities involved in the inequalities are integers, (4) can also be written as follows:

$$0 \leqslant \kappa_1(r-1) \leqslant \kappa_2(r-1) \leqslant \cdots \leqslant \kappa_{r-1}(r-1) \leqslant m-r. \tag{5}$$

Dividing all the terms of (5) by $r-1$, and using the fact that $m = r+\kappa(r-1)$ (which implies that $\kappa = \frac{m-r}{r-1}$), we have

$$0 \leqslant \kappa_1 \leqslant \kappa_2 \leqslant \cdots \leqslant \kappa_{r-1} \leqslant \kappa. \tag{6}$$

Therefore, the number of possible consecutive partitionings is given by the number of ways of choosing a set of $r-1$ integers, $\{\kappa_1, \kappa_2, \ldots, \kappa_{r-1}\}$, from a

set of $\kappa + 1$ values, $\{0, 1, \ldots, \kappa\}$, where repetitions are allowed. This number is given by [27]

$$\binom{\kappa + r - 1}{r - 1}. \tag{7}$$

Since $\kappa = \frac{m-r}{r-1}$, Eq. (7) can be written as follows:

$$\binom{\frac{m-r}{r-1} + r - 1}{r - 1} = \frac{\left(\frac{m-r}{r-1} + r - 1\right)!}{\left(\frac{m-r}{r-1}\right)!(r-1)!}.$$

Hence the theorem. □

Proof of Theorem 2. The theorem follows by a direct consequence of enumerating the possible partitions as detailed in the proof of Theorem 1. It is not included here to avoid repetition. □

Proof of Theorem 3. From the first $r - 1$ **for** loops, we see that the jth **for** loop is executed from κ_{j-1} up to κ. This implies that $\kappa_1 \leqslant \kappa_2 \leqslant \cdots \leqslant \kappa_{r-1}$.

Since the first $r - 1$ **for** loops are executed up to κ, and $\kappa_1 \leqslant \kappa_2 \leqslant \cdots \leqslant \kappa_{r-1}$, the total number of times the last **for** loop is executed is given by the number of ways of selecting a set of integers, $\{\kappa_1, \kappa_2, \ldots, \kappa_{r-1}\}$, from a set of $\kappa + 1$ elements, $\{0, 1, \ldots, \kappa\}$, where $\kappa_1 \leqslant \kappa_2 \leqslant \cdots \leqslant \kappa_{r-1}$, where repetitions are allowed.

Using the result of Theorem 1, this corresponds to the number of $(r-1)$-combinations from a set of $\kappa + 1$ elements, where repetitions are allowed. This number is given by

$$\binom{\kappa + r - 1}{r - 1} = \frac{(\kappa + r - 1)!}{\kappa!(r-1)!}. \tag{8}$$

Since the last **for** loop involves r steps, the total number of steps required by Algorithm **All_Consecutive_Partitionings** is given by

$$r\frac{(\kappa + r - 1)!}{\kappa!(r-1)!} = r\frac{(\kappa + r - 1)(\kappa + r - 2)\cdots(\kappa + 1)\kappa!}{\kappa!(r-1)!} \tag{9}$$

$$= \frac{A_0 + A_1\kappa + \cdots + A_{r-2}\kappa^{r-2} + A_{r-1}\kappa^{r-1}}{(r-1)!}, \tag{10}$$

where $A_0 = r(r-1)!$, $A_{r-1} = r$, $A_i = f_i(r)$ for $i = 1, \ldots, r-2$, and $f_i(\cdot)$ is some well-defined function of r, but which does not contain m.

Eq. (10) can also be written as follows:

$$r + \frac{A_1}{(r-2)!}\kappa + \cdots + \frac{A_{r-2}}{(r-2)!}\kappa^{r-2} + \frac{1}{(r-2)!}\kappa^{r-1}. \tag{11}$$

Since $m = r + \kappa(r-1)$, we have $\kappa = \frac{m-r}{r-1}$. Substituting κ for $\frac{m-r}{r-1}$ in (11), we have:

$$r + \frac{A_1}{(r-2)!}\frac{m-r}{r-1} + \cdots + \frac{A_{r-2}}{(r-2)!}\left(\frac{m-r}{r-1}\right)^{r-2} + \frac{1}{(r-2)!}\left(\frac{m-r}{r-1}\right)^{r-1}. \tag{12}$$

After expanding all the terms, $(\frac{m-r}{r-1})^i$, for $i = 1, \ldots, r-1$, the highest exponent of m is of order $r-1$, which results from expanding $(\frac{m-r}{r-1})^{r-1}$. Then, (12) can be written as follows:

$$B_0 + B_1 m + \cdots + B_{r-2} m^{r-2} + B_{r-1} m^{r-1}, \tag{13}$$

where $B_i = g_i(r)$, for $i = 0, \ldots, r-1$, and $g_i(\cdot)$ is a well-defined function of r but not of m, and in particular, $B_{r-1} = \frac{1}{(r-2)!(r-1)^{r-1}}$. Indeed, for a fixed value of r, B_{r-1} is a constant, and hence (13) reduces to $O(m^{r-1})$. This proves Result (i).

We now consider the case when $m = r$ (or equivalently when $\kappa = 0$). Substituting κ for 0 and m for r in (8), we have:

$$m\binom{m-1}{m-1} = m = O(m),$$

and (ii) is satisfied.

Hence the theorem. □

Proof of Theorem 4. We prove the result by invoking a double induction. We perform an induction on the number of encoding steps, assuming that $x[k]$ is encoded at step k, and for each encoding step, we perform an induction on the number of partitioning steps 'j'.

At the kth encoding step, both the encoder and the decoder can be perceived to be composed of two phases:

(i) Task of Encoder: Encode $x[k]$ using $\mathscr{S}(k)$ and $\mathscr{P}(k)$.Task of Decoder: Decode the relevant sequence into $x^{\star}[k]$ using $\mathscr{S}(k)$ and $\mathscr{P}(k)$.
This part is proved by induction on the number of partitioning steps 'j'.

(ii) Task of both Encoder and Decoder: Update $\mathscr{S}(k)$ and $\mathscr{P}(k)$ to yield $\mathscr{S}(k+1)$ and $\mathscr{P}(k+1)$. The steps involved in both the encoder and decoder are identical.

The inductive proof based on the number of encoding/decoding steps follows.

Basis step: Both the encoder and the decoder start with the same lists of symbols and probabilities, $\mathscr{S}(1)$ and $\mathscr{P}(1)$, where $p_i = 1$, for $i = 1, \ldots, m$. We have to prove that $x^{\star}[1] = x[1]$.

We achieve this part of the proof using an induction on the number of partitioning steps 'j'. We prove that, for every $j \geqslant 1$, the encoder and the decoder obtain the same values of p, t, and b.

Basis step for $x[1]$: We first prove that the encoder and the decoder obtain the same values of p, t, and b, for $j = 1$.

By initialization, both the encoder and the decoder start with the same values of p, t, and b. In the encoder, the main while loop continues until the input symbol, s, is found. This happens when the source symbol $x[k]$ matches s_i. The code alphabet symbol $a_{|\mathscr{A}|-r+1}$ is sent to the output, which is received by the decoder. The main while loop in the partitioning procedure invoked by the decoder is executed, which ends when $a_{|\mathscr{A}|-r+1}$ matches s. This thus implies that when the search ends, p, t, and b have the same values for both the encoder and the decoder.

This proves the basis step because both achieve identical values of p, t and b, implying that they both have identical sublists for the next step, and thus by working in synchronization both achieve encoding and decoding losslessly.

Inductive hypothesis for $x[1]$: We assume that the result holds for $j = n$. This implies that after the nth partitioning step, the encoder and the decoder obtain the same values of p, t, and b.

Inductive step for $x[1]$: We now have to prove that the result holds of $j = n + 1$. From the inductive hypothesis, it is clear that at the $(n+1)$th partitioning step, in both the encoder and the decoder, p, t and b have the same initial values. As in the basis step, the main while loop in the partitioning procedure, when invoked from the encoder, stops when $x[k]$ matches s_i. At this point, $a_{|\mathscr{A}|-r+1}$ is sent to the output, which is received by the decoder. On the other hand, the main while loop at the decoder ends when $a_{|\mathscr{A}|-r+1}$ matches s, thus, terminating the loop. Consequently, in the encoder and the decoder, p, t, and b have the same values.

Again, since both the encoder and the decoder have identical values of p, t and b, this implies that they both have identical sublists for the next step, and achieve encoding and decoding losslessly. The induction thus follows for $x[1]$. In terms of clarification, we see that the encoder and the decoder find the same values of p, t, and b for every partitioning step j, where $j \geqslant 1$. In the encoder, the partitioning stops when $b \leqslant t$ in the encoder. This will happens when there is only one element in $\mathscr{S}(1)$, $x[1]$, determined by $t = b$. In the decoder, when $t = b$, which from the induction on the number of partitioning steps occurs at the same 'j' for both the encoder and the decoder, $s_t(1)$ is recovered. Since t has the same value for the encoder and the decoder, $s_t(1) = x^{\dot{\varsigma}}[1] = x[1]$. This completes the basis step for the induction on the number of encoding steps 'k'.

On the other hand, both the encoder and the decoder increment the weight of $x[1]$ and update $\mathscr{S}(1)$ and $\mathscr{P}(1)$ by swapping $x[1]$ with the top-most symbol in $\mathscr{S}(1)$ whose weight is less than that of $x[1]$, resulting in the same lists, $\mathscr{S}(2)$ and $\mathscr{P}(2)$, for both the encoder and the decoder.

We now continue the outer level of the induction—namely the processing of $x[v]$.

Inductive hypothesis for $x[v]$: We now assume that lossless compression is achieved for a particular number of encoding steps, $k = v$, where $v > 1$. This means that the encoder encodes $x[v]$ using $\mathscr{S}(v)$ and $\mathscr{P}(v)$, and the decoder successfully obtains $x^{\star}[v] = x[v]$. They also update $\mathscr{S}(v)$ and $\mathscr{P}(v)$ using identical rules, obtaining the same symbol and probability lists, $\mathscr{S}(v+1)$ and $\mathscr{P}(v+1)$.

Inductive step: We now have to prove that lossless compression is achieved for $k = v + 1$.

From the inductive hypothesis, we see that:

(a) The encoder and the decoder obtain the same lists $\mathscr{S}(v+1)$ and $\mathscr{P}(v+1)$.
(b) $x^{\star}[1] = x[1], \ldots, x^{\star}[v] = x[v]$.

We have to prove that $x^{\star}[v+1] = x[v+1]$. We prove *this* by an induction on the number of partitioning steps 'j' for this symbol.

Basis step for $x[v+1]$: In this case, we will see that the encoder and the decoder obtain the same values of p, t, and b, for $j = 1$.

From the inductive hypothesis of the induction on the number of encoding steps, $k = v$, we see that the encoder and the decoder start with the same values for p, t, and b. These are $\sum_{i=1}^{m} p_i$, 1, and m respectively.

The main while loop of the partitioning procedure invoked by the encoder ends when the source symbol $x[k]$ matches s_i, sending $a_{|\mathscr{A}|-r+1}$ to the output. The decoder receives $a_{|\mathscr{A}|-r+1}$, and initiates the while loop, which is executed until $a_{|\mathscr{A}|-r+1}$ matches s. At this point, both the encoder and the decoder, assign the same values to p and b, namely p_{temp} and $i-1$ respectively, while t preserves its previous value.

Since both the encoder and the decoder achieve identical values for p, t and b, they maintain identical sublists for the next steps, and thus achieve lossless compression and decompression. The basis step thus follows.

Inductive hypothesis for $x[v+1]$: We assume that the result holds for $j = n$, where j is the number of partitionings, and $n > 1$. This implies that p, b, and t have the same values at the nth partitioning step invoked by the encoder and the decoder.

Inductive step: We now have to prove that the encoder and the decoder have the same values of p, t and b, for $j = n + 1$.

At the $(n+1)$th partitioning step, in both the encoder and the decoder, p, t and b have the same initial values—those values resulting from the nth partitioning step, as enforced by the inductive hypothesis. The main while loop of the partitioning procedure invoked from the encoder stops when the $x[k]$ matches s_i, and $a_{|\mathscr{A}|-r+1}$ is sent to the output. The latter is received by the

decoder, which invokes the $(n+1)$th partitioning step. At this point, the main while loop is executed, which ends when $a_{|\mathscr{A}|-r+1}$ matches s. Consequently, both the encoder and decoder assign p_{temp} and $i-1$ to p and b respectively, while the value of t is maintained from the previous search step.

This completes the induction on the number of partitioning steps 'j' for $x[v+1]$ because, for every $j \geqslant 1$, the encoder and the decoder obtain the same values of p, t and b. They, thus, maintain identical sublists and consequently achieve lossless compression.

The partitioning is performed in the encoder until $b \leqslant t$. This will happen in the decoder when $t = b$, and $s_t(v+1)$ is recovered. Since t has the same value in both the encoder and the decoder, the symbol recovered is $s_t(v+1) = x^{\star}[v+1] = x[v+1]$.

This completes the induction on the number of encoding steps, i.e. the number of symbols encoded and decoded. This result holds for all $v \geqslant 1$, and the theorem is proved. □

Proof of Theorem 5. From the inductive proof of Theorem 4 on the number of encoding steps, we see that for all k, $k = 1, \ldots, M$, $x[k]$ is encoded into an r-ary sequence $y[j] \ldots y[j+\ell_k]$, where ℓ_k is the length of the code word of $x[k]$. This r-ary sequence is simultaneously decoded into $x[k]$. Moreover, each code alphabet symbol, $y[j]$, sent to the output is immediately processed by the decoder in a prefix manner. Consequently, encoding by using Algorithm **Greedy_Adaptive_Fano_Encoding** and decoding by using Algorithm **Greedy_Adaptive_Fano_Decoding** achieve *on-line compression*. □

References

[1] D. Huffman, A method for the construction of minimum redundancy codes, Proceedings of IRE 40 (9) (1952) 1098–1101.
[2] L. Rueda, B.J. Oommen, A nearly optimal Fano-based coding algorithm, Information Processing & Management 40 (2) (2004) 257–268.
[3] D. Hankerson, G. Harris, P. Johnson Jr., Introduction to Information Theory and Data Compression, CRC Press, 1998.
[4] A. Moffat, An improved data structure for cumulative probability tables, Software—Practice and Experience 29 (7) (1999) 647–659.
[5] N. Faller, An adaptive system for data compression, in: Seventh Asilomar Conference on Circuits, Systems, and Computers, 1973, pp. 593–597.
[6] R. Gallager, Variations on a theme by Huffman, IEEE Transactions on Information Theory 24 (6) (1978) 668–674.
[7] D. Knuth, Dynamic Huffman coding, Journal of Algorithms 6 (1985) 163–180.
[8] J. Vitter, Design and analysis of dynamic Huffman codes, Journal of the ACM 34 (4) (1987) 825–845.
[9] J. Muramatsu, On the performance of recency rank and block sorting universal lossless data compression algorithms, IEEE Transactions on Information Theory 48 (9) (2002) 2621–2625.

[10] M.J. Weinberger, E. Ordentlich, On delayed prediction of individual sequences, IEEE Transactions on Information Theory 48 (7) (2002) 1959–1976.
[11] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343.
[12] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Transactions on Information Theory 25 (5) (1978) 530–536.
[13] J. Cleary, I. Witten, Data compression using adaptive coding and partial string matching, IEEE Transactions on Communications 32 (4) (1984) 396–402.
[14] P. Jacquet, W. Szpankowski, I. Apostol, A universal predictor based on pattern matching, IEEE Transactions on Information Theory 48 (6) (2002) 1462–1472.
[15] M. Burrows, D. Wheeler, A block-sorting lossless data compression algorithm, Technical Report SRC-124, Digital Systems Research Center, Palo Alto, CA, May 1994.
[16] M. Effros, K. Visweswariah, S. Kulkarni, S. Verdú, Universal lossless source coding with the burrows wheeler transform, IEEE Transactions on Information Theory 48 (5) (2002) 1061–1081.
[17] J.C. Kieffer, E. Yang, Grammar-based codes: a new class of universal lossless source codes, IEEE Transactions on Information Theory 46 (3) (2000) 737–754.
[18] M. Pinho, W. Finamore, Context-based LZW encoder, IEE Electronic Letters 38 (2002) 1172–1174.
[19] T. Welch, A technique for high performance data compression, IEEE Computer 16 (6) (1984) 8–19.
[20] D. Shkarin, PPM: one step to practicality, in: Data Compression Conference (DCC'02), 2002, pp. 202–211.
[21] W.J. Teahan, D.J. Harper, Combining PPM models using a text mining approach, in: Data Compression Conference (DCC'01), 2001, pp. 153–162.
[22] J. Storer, Data Compression: Methods and Theory, Computer Science Press, 1988.
[23] L. Rueda, Advances in data compression and pattern recognition. Ph.D. Thesis, School of Computer Science, Carleton University, Ottawa, Canada, April 2002. Available from: <http://www.cs.uwindsor.ca/~lrueda/papers/PhdThesis.pdf>.
[24] K. Sayood, Introduction to Data Compression, second ed., Morgan Kaufmann, 2000.
[25] C. Ding, T. Klore, F. Sica, Two classes of ternary codes and their weight distributions, Discrete and Applied Mathematics 111 (2001) 37–53.
[26] R. Arnold, T. Bell, A corpus for the evaluation of lossless compression algorithms, in: Proceedings, Data Compression Conference, Los Alamitos, CA, IEEE Computer Society Press, 1997, pp. 201–210.
[27] K. Rosen, Discrete Mathematics and its Applications, fifth ed., McGraw-Hill, 2002.