An Improved Data Structure for Cumulative Probability Tables

Alistair Moffat*

February 1999

Abstract In 1994 Peter Fenwick at the University of Auckland devised an elegant mechanism for tracking the cumulative symbol frequency counts that are required for adaptive arithmetic coding. His structure spends $O(\log n)$ time per update when processing the sth symbol in an alphabet of n symbols. In this note we propose a small but significant alteration to this mechanism, and reduce the running time to $O(\log(1+s))$ time per update. If a probability-sorted alphabet is maintained, so that symbol s in the alphabet is the sth most frequent, the cost of processing each symbol is then linear in the number of bits produced by the arithmetic coder.

Keywords Compression, adaptive coding, dynamic coding, arithmetic coding, data structure.

Introduction

Arithmetic coding is an important technique in many data compression systems[1–4]. It has the advantage of giving compression close to the source entropy, even when the probability distribution is very skew. Arithmetic coding is also economical in terms of resources when adaptive coding is required. In an adaptive compression model the probabilities of the symbols are updated after each symbol is transmitted, which confers the important benefit of one-pass compression. Minimum-redundancy (or Huffman) coding is the other main coding technique employed in compression systems. It is extremely fast for static coding[5], but the various adaptive coding schemes that have been described to date[6–10] have large constant factors for both memory and processing time[11]. For this reason arithmetic coding is the method of

*Department of Computer Science and Software Engineering, The University of Melbourne, Parkville, Australia 3052. Facsimile: +61 3 93481184. Internet: alistair@cs.mu.oz.au.

choice for on-line adaptive coding. For highly skew probability distributions it also has the advantage of better compression effectiveness.

To implement adaptive arithmetic coding a *statistics* data structure is required. Each coding operation can be described in terms of a triple of parameters (l_s, h_s, t) (see Moffat et al.[12] for details of this process), where

$$\begin{aligned} l_s &= \sum_{i=1}^{s-1} c_i \ , \\ h_s &= \sum_{i=1}^{s} c_i \quad = \quad l_s + c_s \ , \\ t &= \sum_{i=1}^{n} c_i \ , \end{aligned}$$

and where there are n symbols in the alphabet, the ith symbol has occurred c_i times, and the sth symbol is the one to be coded. It is also assumed that p_s, the probability of that symbol, is to be estimated by $p_s = c_s/t$. The statistics data structure is required to track the symbol frequencies c_i and be able to quickly calculate l_s and h_s given a symbol identifier s.

In their landmark implementation of arithmetic coding Witten et al.[2] made use of an array of cumulative frequencies to store the statistics for a character-based model. They also suggested that the symbols in the alphabet be probability-ordered, so that symbol s can be considered to be the sth most frequent. This structure takes $3n$ words of storage for an alphabet of n symbols, and $O(1)$ time in the encoder and $O(\log n)$ time in the decoder to calculate the coding parameters of the sth most frequent symbol. A further $O(s)$ time is then required in each to adapt the statistics—to add one to c_s. The array structure is reasonably efficient when the alphabet is small, as was the case in their illustrative character-based compression experiments. However when the alphabet is large, the $O(s)$ linear-time costs dominate. For example, in word-based compression the alphabet contains many thousands of symbols, yet each symbol is still coded in around 10–12 bits. In such cases the cost of maintaining the statistics structure might be many times greater than the cost of doing the coding itself.

Moffat[13] recognised this state of affairs, and described an improved statistics structure that calculates and then updates the coding parameters for symbol s in time $O(\log(s+1))$. Moffat further noted that $\log_2 s \le \log_2(1/p_s)$, and hence that the time taken is proportional to the number of bits output when a symbol of probability p_s is coded. That is, the cost of manipulating the statistics structure is linear in the number of bits produced by the coder. The drawback of the improved mechanism is that it uses $4n$ words of memory on an n symbol alphabet, which

might be a prohibitive overhead.

Fenwick[14] invented a simpler method that is not as efficient asymptotically, but is as fast in practice, and has as a single crucial advantage that it requires just n words of storage, less even than the Witten et al.[2] approach. Using the Fenwick structure all operations can be carried out in $O(\log n)$ time. In this note we modify the structure of Fenwick and develop a mechanism that performs the required computations in $O(\log s)$ time* and n words of space. That is, we blend the observations of Moffat and Fenwick and obtain a structure superior to both.

Fenwick's Structure

The operations that must be supported by an arithmetic coding statistics structure are:

get_lower_bound(s)

Given a symbol identifier s, return the cumulative lower bound l_s.

get_count(s)

Given a symbol identifier s, return the occurrence frequency c_s.

get_symbol$(target)$

Given an integer *target* (see Moffat et al.[12] for a detailed description) such that $0 \le target < t$, return s such that $l_s \le target < h_s$.

increment_count(s)

Add one to the occurrence frequency c_s of symbol s.

scale_all_counts$()$

Scale the set of frequency counts, replacing c_s by $\lceil c_s/K \rceil$ for some constant K, usually taken to be $K = 2$.

The encoder makes calls to *get_lower_bound*() and *get_count*() to establish the coding triple (l_s, h_s, t) for each symbol, and then uses *increment_count*() to update the statistics. The decoder calls *get_symbol*() to determine the symbol being decoded, and then uses *get_lower_bound*(), *get_count*(), and *increment_count*() to mimic the actions of the encoder. Both—in synchronisation—make occasional calls to *scale_all_counts*().

It should be noted that while these operations have been described as if the stream being compressed is of a single type of symbol, in most practical compression systems several or many coding contexts are employed, and each symbol is coded

*It is convenient to use $O(\log s)$ as an abbreviation for $O(\log(s+1))$; the exact expression $\log(1+s)$ will be used where necessary.

with respect to the statistics accumulated for just one of the possible contexts. For example, in a context-based compression system a conditioning class might be established for each possible immediately preceding character, so that after a letter "q" different probabilities are used than after say the letter "t". Hence, each of the listed statistics operations should also be assumed to have an extra parameter specifying the context to be used. For simplicity that parameter is omitted in the discussion here, and we suppose that a single context, and the symbols to be coded in that context, has been isolated.

Let us now turn to the structure devised by Fenwick[14]. Define $size(s)$ for positive integer s to be the largest power of two that evenly divides s,

$$size(s) = \max\{2^v \mid s \bmod 2^v = 0 \text{ for } v \in \{0, 1, 2, \ldots\}\} \ ,$$

and, based upon $size$, define two more functions, $forward()$ and $backward()$,

$$\begin{aligned} forward(s) &= s + size(s) \\ backward(s) &= s - size(s) \ . \end{aligned}$$

The essence of Fenwick's proposal is an n-array F that stores partial sums of symbol frequencies. With c_s the observed frequency of symbol s, the array F satisfies the invariant

$$F[s] = \sum_{i=backward(s)+1}^{s} c_i \ .$$

Figure 1 gives an example of the values stored in array F for a small alphabet of $n = 9$ symbols. The first row—marked (a)—lists the actual c_s values that are assumed to have been observed. The l_s and h_s coding bounds for each of these symbols are shown in the row marked (b), and the $backward()$ and $forward()$ functions in row (c). The row marked (d) shows the sets of c_i values that combine to give each element of F, and row (e) lists the partial cumulative sums stored in array F.

To use the array F to calculate the coding parameters l_s and h_s for some symbol s, observe that by definition $h_s = F[s] + h_{backward(s)}$ holds, and hence recursively that

$$h_s = F[s] + F[backward(s)] + F[backward(backward(s))] + \cdots \ ,$$

with the summation continuing until a point is reached at which $F[0]$ would be required. That is, a value for h_s can be calculated with a simple loop that sums the appropriate values in the F array, starting at s and moving *backward* in ever-increasing leaps. The value of $l_s = h_{s-1}$ can be similarly calculated by a computation that commences at $F[s-1]$, or can be calculated more directly by an alternative mechanism[14].

The time taken is logarithmic in s. To see this, note that

$$size(backward(s)) \geq 2 \cdot size(s),$$

and hence that each movement backwards from a starting point s covers at least double the distance of the previous backwards movement. This analysis is, of course, based upon the

location s	1	2	3	4	5	6	7	8	9
(a) symbol frequencies									
c_s	15	10	8	5	5	4	4	2	1
(b) corresponding bounds									
l_s	0	15	25	33	38	43	47	51	53
h_s	15	25	33	38	43	47	51	53	54
(c) aggregation functions									
$backward(s)$	0	0	2	0	4	4	6	0	8
$forward(s)$	2	4	4	8	6	8	8	16	10
(d) range of c_i values summed in $F[s]$									
	1–1	1–2	3–3	1–4	5–5	5–6	7–7	1–8	9–9
(e) value stored in $F[s]$									
$F[s]$	15	25	8	38	5	9	4	53	1
(f) revised values in array F after $increment\_count(3)$									
$F[s]$	15	25	**9**	**39**	5	9	4	**54**	1

Figure 1: Example of the use of Fenwick's cumulative frequency data structure.

assumption that evaluation of the function *size*() requires $O(1)$ time per call. How this is achieved is described below.

Now consider how c_s might be increased by one. A similar loop is used, starting at s. This time, however, the loop moves to the right, since c_s is a component of the partial sums stored at $F[s]$, $F[forward(s)]$, and so on. Since the *size* of each element accessed is at least double the *size* of the preceding element accessed, the number of calls to the function *size* is again at most logarithmic, this time in $n - s$. Row (f) of Figure 1 shows the result of an *increment_count*(3) function call. The values changed as a result of this operation are marked in boldface.

The decoder uses the same F array to search for the specified *target*. The search commences at the last power of two less than or equal to n, the size of the alphabet. Denote this value $m = 2^{\lfloor \log_2 n \rfloor}$; in the example, $m = 8$. If $F[m] \leq target$, then the first m positions of F can be stepped over, and the search restricted to the values $s \geq m$. The *target* is then decremented by $F[m]$, and the search continued recursively from $F[m + m/2]$. A similar process—without the decrementing of the *target*—is followed in the first m positions of F if $F[m] > target$. The search terminates when m—which halves at each iteration—reaches zero, and so is logarithmic in n. Moffat et al.[12] give pseudo-code for the *get_lower_bound*() and *get_symbol*() operations.

The *get_lower_bound*() and *increment_count*() procedures depend on the function *size*(). If two's complement arithmetic is being used *size*() can be calculated very simply[14] using

$$size(s) = s \text{ AND } (-s) \ ,$$

where AND denotes a bit-by-bit logical "and" operation on two values. For example in a four bit integer representation, when $s = 6 = 0110_2$, the value -6 is represented by the bit pattern 1010_2, and the conjunction of these two bit strings yields $0010_2 = 2 = size(6)$. That is, *size*() can be calculated in $O(1)$ time per call.†

Overall, each symbol handled by the encoder requires $O(\log s)$ time for the calculation of the bounds, and $O(\log(n - s))$ time for frequency update. Irrespective of the value of s, the sum of these is $O(\log n)$. The decoder requires $O(\log n)$ time for both the searching phase and the consequent increment operation.

A Small Reorganisation

Suppose instead that a different set of cumulative sums is maintained in an array M, going forwards rather than backwards:

$$M[s] = \sum_{i=s}^{forward(s)-1} c_i \ .$$

†Indeed, as was noted by one of the referees, *size*() can be calculated in all binary integer representations as s AND $(2^N - s)$ for any value N for which $2^N > n$, a calculation which makes use of only positive integers and so is independent of the particular representation used for negative values.

Figure 2 shows the revised arrangement, using the same set of symbol frequencies as in Figure 1. Row (a) shows the symbol frequencies; row (b) the bounds; and row (c) the range of symbol frequencies stored in each position of the M array.

The access operations must be similarly reorganised. Figure 3 gives details of the three operations required in the encoder. To calculate the lower bound l_s for a symbol s two different components are computed. The first, at step 2 of function *get_lower_bound*(), is a sum of powers of two to the left of s, namely $M[1] + M[2] + M[4] + \cdots + M[2^{\lceil \log_2 s \rceil - 1}]$. At the completion of this loop $p = 2^{\lceil \log_2 s \rceil}$, and the sum in l includes all of the c_i values up to s and beyond s through to but not including p, the next power of two greater than or equal to s. That is, variable l is greater than or equal to l_s and is equal only if s is a power of two.

The second calculation, described at step 4, computes the excess that must be subtracted from l to account for symbols known to lie to the right of s. It is calculated by stepping forwards from s, stopping when p is reached. Each of these two loops requires $O(\log s)$ iterations when processing the sth symbol of the alphabet. Figure 3 also shows, in function *get_count*(), the process used to calculate the frequency count of a symbol s so that h_s can be calculated. This function requires $O(1)$ time on average, since half of the time no iterations of the loop at step 2 are required, a quarter of the time there is just one iteration, and so on, assuming that each position in M is equally likely to be the subject of such an enquiry.

Figure 3 also shows the process whereby the frequency count for a symbol is incremented. The *backward*() links are used. Row (e) of Figure 2 shows in bold-face the values in array M that are affected when *increment_count*(3) is executed. Now the altered values all lie to the left of s rather than to the right, as was the case in row (e) in Figure 1, and the time taken is again $O(\log s)$, where s is the index in M of the value being adjusted.

The function *get_symbol*() undergoes a similar reorganisation, and is shown in Figure 4. The search proceeds outwards from $p = 1$, doubling the index p at each stage. Similar search strategies in other applications have been described by Elias[15] and Bentley and Yao[16]. The location of the symbol s that was coded onto the *target* is then determined by a converging search, in much the same way as was done in the original Fenwick structure, but the search is constrained to the interval $p \le s < 2p$, and the cost of this phase is also $O(\log s)$. Variable e records the excess weight in $M[s]$ that has already been discounted during the narrowing search, and is set to zero each time the search moves rightward.

At the same time as the symbol number s is being determined in function *get_symbol*() the effect of the calls to *get_lower_bound*() and *increment_count*() can be incorporated, since all of the array locations required in those two functions are visited during the two *get_symbol*() loops. That is, in an actual implementation of the data structure only the encoder makes explicit calls to *get_lower_bound*() and *increment_count*(), and in the decoder *get_symbol_and_increment_count*() is an omnibus function somewhat more powerful than the functionality that was sketched earlier. The same extended functionality can also be employed in the Fenwick organisation.

The other important operation is that of count scaling—approximately halving each of the frequency counts in the structure so as to allow t, the sum of the frequencies c_s, to

be bounded. Figure 5 shows $O(n)$-time functions to convert the array M from a simple list of symbol frequencies to the cumulative structure (function *counts_to_cumulative*()), and, as the inverse transformation, from the cumulative structure to a list of symbol frequencies (function *cumulative_to_counts*()). In conjunction with a loop that halves each symbol frequency, these two function provide an $O(n)$-time mechanism that *in-situ* scales the whole of the cumulative frequency structure. Fenwick described another approach to count scaling that consists of "adding" $-\lfloor get\_count(s)/2 \rfloor$ to the frequency of each symbol s. This approach has the same effect, but requires $O(n \log n)$ time in total to update the n frequencies, and the mechanism described in Figure 5 (which can be readily modified to apply to the Fenwick organisation) is superior.

Probability-Sorted Alphabets

Since the cost of access in the revised structure is related to the symbol number that is accessed, it is clearly desirable to arrange the alphabet so that the symbols are in decreasing probability order. When so arranged a close relationship exists between $\log s$, the cost of maintaining the statistics for the sth symbol, and $-\log_2 p_s$, the number of bits emitted by the arithmetic coder as a result of that symbol being coded. When symbol s is the sth most frequent in the alphabet, its frequency c_s can be no more than t/s, since there are at least s symbols of this or greater frequency. Furthermore, the probability p_s of symbol s is estimated as c_s/t, so the probability of symbol s is at most $1/s$. Hence, an arithmetic coder must generate at least $-\log_2(1/s) = \log_2 s$ bits when coding this symbol. That is, the cost of maintaining the statistics is never more than one greater than the number of bits produced by the coder. This relationship was noted by Moffat[13].

To ensure that the alphabet is in decreasing frequency order, two permutation arrays are used, *symbol_to_rank*[] and *rank_to_symbol*[]. The process of incrementing the count for a symbol of necessity becomes a little more complex; now the symbol ordering must be permuted if the planned increment to s would disturb the requirement that the array M be in non-increasing frequency order. Prior to an actual increment operation on symbol s the symbol s' that is the least ranked symbol with the same frequency as s is identified, and the two permutation arrays *symbol_to_rank*[] and *rank_to_symbol*[] adjusted so as to exchange the relative positions of s and s'. Then, and only then, is it appropriate to increment s.

The cost of this adjustment is never more than $O(\log s)$, since binary search on the domain $1 \ldots s$ will find s'. On average the cost is very small indeed: most of the time the symbol ordering will not need to be adjusted, and so an exponential and binary search leftward from s will find s' even more quickly than a binary search on the full array, and have a maximum cost that is again logarithmic in s.

There are three drawbacks of this additional structure. The first and most obvious is that $2n$ words of extra space are required, a substantial imposition when the main structure requires just n words.

The second drawback is more subtle, and is that increments must now always be by unit amounts. The function *increment_count*() in Figure 3 can actually cope with any alteration to the frequency c_s, whether positive or negative, and by any amount, integer or fractional.

But if the frequencies are to be maintained in decreasing order the process of exchanging s and the leftmost value s' of the same frequency require that after s has been adjusted it is in the right spot, which can only be guaranteed if the increment is by a standard unit amount. This then precludes the possibility of some symbols being incremented by non-unit amounts when they are coded.

The third drawback is that use of an omnibus decoding function of the kind illustrated in Figure 4 is not possible, as cumulative frequencies along the search path cannot be adjusted until it is known whether or not the symbol will remain in the same position of the array M.

Experimental Results

Let us now compare the costs of the Fenwick and new mechanisms more accurately. We do this is two ways: first, by estimating for each the number of loop iterations required for encoding and decoding a symbol s in an alphabet of n symbols; and then by timing actual experimental implementations.

Comparing the two mechanisms by counting loop iterations assumes that each loop iteration requires the same amount of computation, and, roughly speaking, this is a valid assumption. To estimate the number of iterations, observe that the loop in function *get_lower_bound*() associated with Fenwick's structure (this functions is not detailed in this paper) iterates once for each 1-bit in the integer description of s. Making the assumption that bits to the right of the leading one bit are equally likely to be ones and zeroes then means that the average number of loop iterations required in this function is approximately $0.5 \log_2 s$. The column headed "Encoding Fenwick" in Table 1 shows this cost.

Function *increment_count*() then iterates once for each zero bit to the left of the rightmost (least significant) one bit in s, up to a maximum of $\log_2 n$ times. Since there are $\log_2 n - \log_2 s$ zero-bits to the left of the most significant one-bit, and, by the same randomness assumption, $0.5 \log_2 s$ zero-bits between the leftmost and rightmost one-bits of s, there are a further $\log_2 n - 0.5 \log_2 s$ loop iterations associated with function *increment_count*().

In the revised structure the first loop in function *get_lower_bound*() (step 2 in Figure 3) iterates exactly $\lfloor \log_2 s \rfloor$ times, and the second loop (step 4) iterates once for each 1 bit in a number of the same magnitude as s. The structure of this second loop is then repeated in function *increment_count*(), going backward rather than forward. Making the same assumption about the randomness of one and zero bits, the total encoding cost of the revised structure is thus $2.0 \log_2 s$ loop iterations. Table 1 also lists the cost of the two alternative decoding structures, in both cases assuming omnibus *get_symbol_and_increment_count*() functions of the form shown in Figure 4.

Comparing the various expressions listed in Table 1, the number of loop iterations for the proposed variant is less than for Fenwick's mechanism when $2 \log_2 s < \log_2 n$, that is, when $s < \sqrt{n}$.

Whether or not these conditions occur in practice, and whether the presumed relativities are correct, is determined with reference to actual data files and an implementation. Table 2 shows representative values of $\log_2 n$ and $\log_2 s$ (averaged over the stream) for three symbol

location s	1	2	3	4	5	6	7	8	9
(a) symbol frequencies									
c_s	15	10	8	5	5	4	4	2	1
(b) corresponding bounds									
l_s	0	15	25	33	38	43	47	51	53
h_s	15	25	33	38	43	47	51	53	54
(c) range of c_i values summed in $M[s]$									
	1–1	2–3	3–3	4–7	5–5	6–7	7–7	8–9	9–9
(d) value stored in $M[s]$									
$M[s]$	15	18	8	18	5	8	4	3	1
(e) revised values in array M after *increment_count*(3)									
$M[s]$	15	**19**	**9**	18	5	8	4	3	1

Figure 2: Example of the use of the improved data structure.

	Encoding		Decoding	
	Fenwick	Improved	Fenwick	Improved
get_lower_bound()	$0.5 \log_2 s$	$1.5 \log_2 s$	—	—
get_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
increment_count()	$\log_2 n - 0.5 \log_2 s$	$0.5 \log_2 s$	—	—
get_symbol_and_ increment_count()	—	—	$1.0 \log_2 n$	$2.0 \log_2 s$

Table 1: Average number of loop iterations to code symbol s in an alphabet of n symbols.

get_lower_bound(s)

/* Return the cumulative frequency of the symbols prior to s in the alphabet */

1. Set $p \leftarrow 1$ and $l \leftarrow 0$
2. While $p < s$ do
   Set $l \leftarrow l + M[p]$ and $p \leftarrow 2p$
3. Set $q \leftarrow s$
4. While $q \neq p$ and $q \leq n$ do
   Set $l \leftarrow l - M[q]$ and $q \leftarrow \mathit{forward}(q)$
5. Return l

get_count(s)

/* Return the frequency count of symbol s */

1. Set $c \leftarrow M[s]$ and $q \leftarrow s + 1$ and $z \leftarrow \min(\mathit{forward}(s), n + 1)$
2. While $q < z$ do
   Set $c \leftarrow c - M[q]$ and $q \leftarrow \mathit{forward}(q)$
3. Return c

increment_count(s)

/* Add one to the frequency of symbol s */

1. Set $p \leftarrow s$
2. While $p > 0$ do
   Set $M[p] \leftarrow M[p] + 1$ and $p \leftarrow \mathit{backward}(p)$
3. Return

Figure 3: Encoding functions using the improved structure.

get_symbol_and_increment_count(*target*)

/* Returns the symbol number s such that
get_lower_bound$(s) \leq$ *target* $<$ *get_lower_bound*(s) + *get_count*(s).
Also returns the value l_s that would be returned by
get_lower_bound(s) and the value c_s; and adjusts the array M to
allow for one more appearance of symbol s, as if
increment_count(s) had been called */

1. Set $p \leftarrow 1$ and $l \leftarrow 0$
2. While $2p \leq n$ and $M[p] \leq$ *target* do
   Set *target* $\leftarrow$ *target* $- M[p]$ and $l \leftarrow l + M[p]$ and $p \leftarrow 2p$
3. Set $s \leftarrow p$ and $m \leftarrow p/2$ and $e \leftarrow 0$
4. While $m \geq 1$ do
   (a) If $s + m \leq n$ then
   Set $e \leftarrow e + M[s + m]$
   If $(M[s] - e) \leq$ *target* then
   Set *target* $\leftarrow$ *target* $- (M[s] - e)$ and
   $l \leftarrow l + (M[s] - e)$ and
   $M[s] \leftarrow M[s] + 1$ and $s \leftarrow s + m$ and $e \leftarrow 0$
   (b) Set $m \leftarrow m/2$
5. Set $M[s] \leftarrow M[s] + 1$
6. Return s and l and *get_count*$(s) - 1$

Figure 4: Decoding using the improved structure, with symbol frequency update incorporated.

scale_counts()

/* Approximately halve each of the n frequencies represented by array M */

1. *cumulative_to_counts*(n)
2. For $s \leftarrow 1$ to n do
   Set $M[s] \leftarrow (M[s] + 1)$ div 2
3. *counts_to_cumulative*()

cumulative_to_counts()

/* Convert the array M from a cumulative frequency structure into a simple array of symbol frequencies */

1. Set $p \leftarrow 2^{\lfloor \log_2 n \rfloor}$
2. While $p > 1$ do
   (a) Set $s \leftarrow p$
   (b) While $s + p/2 \leq n$ do
   Set $M[s] \leftarrow M[s] - M[s + p/2]$
   Set $s \leftarrow s + p$
   (c) Set $p \leftarrow p/2$

counts_to_cumulative()

/* Convert the array M from a simple array of symbol frequencies into a cumulative frequency structure */

1. Set $p \leftarrow 2$
2. While $p \leq n$ do
   (a) Set $s \leftarrow p$
   (b) While $s + p/2 \leq n$ do
   Set $M[s] \leftarrow M[s] + M[s + p/2]$
   Set $s \leftarrow s + p$
   (c) Set $p \leftarrow 2p$

Figure 5: Scaling frequency counts.

	Characters	*Words*	*Non-words*
Length of message	279,534,950	85,983,232	85,983,232
n	255	287,587	8,314
Entropy (bits per symbol)	4.84	11.17	2.46
Average $\log_2 n$	7.99	18.13	13.02
Average $\log_2(1+s)$ (unsorted)	6.31	9.71	3.38
Average $\log_2(1+s)$ (sorted)	3.00	7.81	1.72

Table 2: Statistics for three test messages.

distributions extracted from a file of approximately 250 MB of English text, including SGML markup, taken from the *Wall Street Journal*. The three columns correspond to symbol streams generated by considering the words of the text, the non-word strings in the text (Moffat et al.[12] describe the word-based model in more detail), and the characters of the text. In the case of the files *Words* and *Non-words* the symbols are ordinal numbers assigned in order of first appearance, starting with 1 for the first symbol in each stream. In the file *Characters*, the ASCII codes of characters are used to represent themselves, and so most values lie between about 30 and 128.[‡] For example, the text makes use of 287,587 distinct word codes. In total about 86,000,000 word and non-word codes appear in the symbol streams considered, and about 280 million character codes.

For the word codes, the average value of $\log_2(1+s)$ of 9.71 is approximately half $\log_2 n$, and so the modified statistics structure may result in faster processing than the original Fenwick arrangement. The expected benefit for file *Non-words* is even greater. The non-random pattern in these two files is a natural consequence of the manner in which symbol identifiers were assigned. The symbols which appear early in streams such as these are likely to appear often, and so allocating low symbol numbers to them is beneficial. If the underlying probability distribution is then sufficiently non-uniform, fast data structure manipulation is possible. On the other hand, the additional advantage accruing through sorting these two alphabets is small, and may not be warranted. The use of the permutation arrays reduces the number of loop iterations in the various statistics functions, but introduces additional costs because of array indirections, and because as many as $\log s$ iterations are required in the position-searching loop prior to each call to *increment_count*().

Table 3 lists actual execution times to compress and decompress the three test files on a 266 MHz Intel Pentium II with 256MB RAM and 512kB cache when coupled with the arithmetic coding routines described by Moffat et al.[12], and called from a driver program intended to compress streams of integers of the form described above. The results in the table show that if the alphabet being used has a naturally-occurring probability order (files *Words* and *Non-words*), then the improved structure does yield faster times for both encoding and decoding. On the other hand, explicitly probability-sorting the alphabet through the use of auxiliary arrays is not beneficial, even when there is no natural ordering

[‡] These files have been used for a range of other experimental work, and for technical reasons file *Characters* is prefixed by the bytes from 1 through to 255 inclusive. There are actually 94 distinct characters that appear in the source text.

	Characters		*Words*		*Non-words*	
	Encoding	Decoding	Encoding	Decoding	Encoding	Decoding
Fenwick	33.34	59.38	290.72	432.99	111.76	221.50
Improved	37.35	60.60	258.00	359.17	102.24	182.00
Improved (p)	44.49	61.29	315.64	390.43	129.69	197.78

Table 3: Compression time (CPU seconds) using a 266 MHz Intel Pentium II with 256MB RAM and 512kB cache. Row "Fenwick" shows decoding times for Fenwick's structure when an omnibus function *get_symbol_and_increment_count*() is used; row "Improved" lists times for the improved structure with omnibus decoding but without the use of auxiliary permutation arrays; and row "Improved (p)" shows the time required when auxiliary permutation arrays are used, without omnibus decoding.

(file *Chars*) on the alphabet.

The times listed in Table 3 in the rows "Fenwick" and "Improved" are for omnibus decoding, using functions akin to that of Figure 4. In the case of the Fenwick structure, the omnibus decoder executed slightly faster than an equivalent decoder with separate calls to *get_symbol*() and *increment_count*(). But in the case of the improved decoder, the reverse was the case, possibly as a result of the greater complexity of the decoding loop making it harder for the compiler to optimise it. This anomalous behaviour is a graphic reminder that all experimental results, including those of Table 3, must be interpreted carefully: they apply to one implementation, using one compiler, and executed on one particular architecture.

Summary

A reorganisation of Fenwick's cumulative frequency data structure has been proposed. The revised method allows symbols to be arithmetically encoded and decoded in $O(1 + \log s)$ time, an improvement over the $O(\log n)$ time of the Fenwick mechanism. An efficient technique for count scaling has also been described.

In conjunction with $2n$ words of auxiliary space, the proposed method allows arithmetic encoding and decoding in time asymptotically linear in the number of bits being output and input, improving upon the $3n$ words needed for the method described by Moffat [13]. In practice the cost of the necessary array indirections makes this enhancement of limited practical utility. On the other hand, without the auxiliary arrays the new structure also offers faster processing for certain types of naturally occurring alphabets, in which frequently-occurring symbols are assigned relatively small symbol identifiers, and it is without auxiliary arrays that the improved structure appears to be the most useful in practice.

Acknowledgements

This work was supported by the Australian Research Council. Andrew Turpin (University of Melbourne) carried out the experiments. The anonymous referees and Sascha Kratky (Uni Software, Austria) also provided helpful input.

Source Code

Source code for the improved statistics structure is available at `http://www.cs.mu.oz.au/~alistair/arith_coder/`.

References

[1] G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, March 1984.

[2] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541, June 1987.

[3] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[4] P. G. Howard and J. S. Vitter. Arithmetic coding for data compression. *Proc. IEEE*, 82(6):857–865, June 1994.

[5] A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, October 1997.

[6] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, November 1978.

[7] G. V. Cormack and R. N. Horspool. Algorithms for adaptive Huffman codes. *Information Processing Letters*, 18(3):159–165, March 1984.

[8] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163–180, June 1985.

[9] J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34 (4):825–845, October 1987.

[10] J. S. Vitter. Algorithm 673: Dynamic Huffman coding. *ACM Transactions on Mathematical Software*, 15(2):158–167, June 1989.

[11] A. Moffat, N. Sharman, I. H. Witten, and T. C. Bell. An empirical evaluation of coding methods for multi-symbol alphabets. *Information Processing & Management*, 30(6): 791–804, November 1994.

[12] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, July 1998.

[13] A. Moffat. Linear time adaptive arithmetic coding. *IEEE Transactions on Information Theory*, 36(2):401–406, March 1990.

[14] P. Fenwick. A new data structure for cumulative probability tables. *Software—Practice and Experience*, 24(3):327–336, March 1994. Errata published in 24(7):677, July 1994.

[15] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.

[16] J. Bentley and A. C-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, August 1976.