A New Data Structure for Cumulative Probability Tables : an Improved Frequency-to-Symbol Algorithm.

Peter M. Fenwick

UCSC-CRL-95-17
March 20, 1995

Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand.
peter-f@cs.auckland.ac.nz

ABSTRACT

A recent paper presented a new efficient data structure, the "Binary Indexed Tree", for maintaining the cumulative frequencies for arithmetic data compression. While algorithms were presented for all of the necessary operations, significant problems remained with the method for determining the symbol corresponding to a known frequency. This report corrects that deficiency.

This report is being filed also with the Department of Computer Science, The University of Auckland, as Technical Report 110, ISSN 1173-3500

Keywords: Binary Indexed Tree, arithmetic coding, cumulative frequencies

1. Introduction

In a recent paper the author presented the "Binary Indexed Tree", a new data structure for maintaining the cumulative frequency tables which are an essential part of arithmetic data compression. [1] The new structure is compact and fast, requiring no more space than is required for a simple table of integer frequencies, and with access times proportional to the logarithm of the alphabet size (with a small constant of proportionality.) The frequencies are held in a table with each entry containing the sum of $1, 2, 4, 8, \ldots$ frequencies, depending on the bit pattern of the index. Accessing the structure requires operations based on the least-significant 1-bit of the index, which are easily achieved by combinations of logical and arithmetic operations. The description here assumes a knowledge of the original paper, to which readers are referred.

The paper gave methods for incrementing the frequency of a given symbol, reading the cumulative frequency of a symbol and reading the individual frequency of an element. It also presented an algorithm for determining the symbol corresponding to a given frequency, but one which required all symbols to have non-zero frequencies. This serious limitation was recognised at the time and indeed the comment was made that "There seems to be no efficient programming solution to this problem, but a simple detour is to assume a constant base frequency for all values, adjusting the cumulative or real frequencies as they are read."

[1] P.M. Fenwick, "A new data structure for cumulative frequency tables", *Software – Practice and Experience, Vol 24, No 3, pp 327–336, Mar 1994*

2. The improved algorithm

Experience with the new structure has shown that it works very well for coding, but that the problems in determining the symbol for a given frequency had to be overcome for satisfactory decoding. In particular, high-order coding models require sparse tables with many zero frequencies, which is precisely the case which the original algorithm did not handle. The suggested detour is also quite unsatisfactory. These problems forced the development of the improved frequency to symbol algorithm which follows.

It is possible to use a simple search through the table, using the code in Figure 1 as a direct implementation of the specification of the frequency relationships, but the serial search is clearly inefficient. However this program does serve as a reference against which any better algorithm may be tested, and also emphasises the relationships between the various frequencies, relationships which must be respected in any satisfactory implementation.

```
int referenceGetSymbol(int F)
   {
   int k;
   for (k = 0; k <= TblSize; k++)
     if (F >= getCumFreq(k-1)  &&
         F <  getCumFreq(k-1) + getIndFreq(k))
       return k;
   }
```

Figure 1. A reference version of a "getSymbol" routine

Like the original, the new algorithm uses a form of binary search, exploiting the recursive nature of the data structure. If, for example, we are working with an interval covering a range of 16 elements, we are concerned only with the frequencies within that range. Element 8 contains the sum of the elements 1–8 and testing the probe frequency against it allows us to select the upper half-range or the lower half-range. If in the upper half we adjust its "base" frequency (using the value in element 8) and then move the range to cover the old elements 8–16. If in the lower half we just adjust the range to cover 0–8. The recursive structure allows us to repeat the operation on successively smaller intervals until we converge to a single element.

The code presented later basically follows the above description but with two other complications, both arising because we dealing not with simple values, but with frequencies and ranges of frequencies, with specific interpretations in the context of arithmetic compression.

1. If an element in the table has a cumulative frequency of 5 and an individual frequency of 2, then it represents a range of frequencies f, such that $3 \leq f < 5$. However, a probe frequency $f = 5$ represents a frequency in the range $5 \leq f < 6$, when its following "fractional" bits are allowed for. Thus a probe frequency f corresponds to the interval for a cumulative frequency $f + 1$, and must be incremented by 1 before we start the search. (Remember that arithmetic coding produces a number of arbitrarily high precision and that this number is successively doubled throughout the decoding operation. A few of the more-significant bits appear as the integer value which used here as the probe frequency, but the less significant bits are still there, though hidden, and can represent any value $0 \leq v < 1$.)

2. The second point follows on from the preceding discussion. While we appear to be searching for a value, we are actually searching for an interval and the value we find really belongs to the interval just below the one we want. We must therefore increment the final index as determined by the search. (Referring back to the program of Figure 1, note that when testing for element k, we examine `cumFreq[k-1]`.)

The final code, given in Figure 2, assumes that the structure is held in an integer array `T[Size]`. It is a direct implementation of the algorithm as described, including the special adjustments to the probe frequency and the final symbol index.

```
int getSymbol(int Freq)
    {
    int baseIx, testIx, half;

    if (Freq < T[0])            /* test for within first element */
         return 0;
    baseIx = 0;                 /* initial base of search range */
    Freq -= T[0];               /* subtract root value from freq */
    Freq++;                   /* must probe with (Freq+1) */
    half = Size >> 1;           /* get mid point of valid range */
    while (half > 0)            /* repeat until range disappears */
       {
       testIx = baseIx+half;    /* probe midpoint of range */
       if (Freq > T[testIx])    /* if above mid-point */
          {
          baseIx = testIx;      /* move base to probe index */
          Freq -= T[baseIx];    /* subtract base frequency */
          }
       half >>= 1;              /* halve the gap */
       }
    return baseIx+1;            /* interval above the found value */
    }
```

Figure 2. The revised frequency-to-symbol routine

3. Acknowledgements

This work was supported by grant A18/XXXXX/62090/F3414032 from the University of Auckland and completed while the author was on Study Leave at the University of California – Santa Cruz. The author acknowledges the contributions of both of these institutions.