A New Data Structure for Cumulative Probability Tables

Peter Fenwick,

Technical Report 88

ISSN 1173-3500

28 May 1993

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

Summary

A new method (the "Binary Indexed Tree") is presented for maintaining the cumulative probabilities which are needed to support dynamic arithmetic data compression. It is based on a decomposition of the cumulative probabilities into portions which parallel the binary representation of the index of the table element (or symbol). The operations to traverse the data structure are based on the binary coding of the index. In comparison with previous methods, the Binary Indexed Tree is faster, using more compact data and simpler code. The access time for all operations is either constant or proportional to the logarithm of the table size. In conjunction with the compact data structure, this makes the new method particularly suitable for large symbol alphabets.

Introduction

A major cost in adaptive arithmetic data compression is the maintenance of the table of cumulative probabilities which is needed in reducing the range for successive symbols. Witten, Neal and Cleary [5] ease the problem by providing a move-to-front mapping of the symbols which ensures that the most frequent symbols are kept near the front of the search space. It works well for highly skewed alphabets (which may be expected to compress well) but is much less efficient for more uniform distributions of symbol probability. Moffat [3] describes a tree structure (actually a heap) which provides a linear-time access to all symbols. Jones [2] uses splay trees to provide an optimised data structure for handling the probability tables. The three techniques will be referred in this paper as MTF, HEAP and SPLAY, respectively. In all cases they attempt to keep frequently used symbols in quickly-referenced positions within the data structure, but at the cost of sometimes extensive data re-organisation.

This current paper describes a new method which uses only a single array to store the probabilities, but stores them in carefully chosen pattern to suit a novel search technique whose cost is proportional to the number of 1 bits in the element index. This cost applies to both updating and interrogating the table. In comparison with the other methods it is simple, compact and fast and involves no reorganisation or movement of the data.

Principles

The basic idea is that, just as an integer is the sum of appropriate powers of two, so can a cumulative probability be represented as the appropriate sum of sets of cumulative "sub probabilities". Thus, if the index contains a "2 bit" we include two probabilities, if it has an "8 bit" we include 8 probabilities, and so on. Figure 1 shows a table of size 16.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	0	1	1…2	3	1…4	5	5…6	7	1…8	9	9…10	11	9…12	13	13…14	15
Item Prob	0	2	0	1	1	1	0	4	4	0	1	0	1	2	3	0
Cum Prob	0	2	2	3	4	5	5	9	13	13	14	14	15	17	20	20
Stored Values	0	2	2	1	4	1	1	4	13	0	1	0	2	2	5	0

Figure 1. Example of the table

The first row is simply the index. The second shows the contents of that entry of the table; for example, element 4 contains the sum of elements 1 to 4 inclusive, while element 6 has the sum of element 5 and element 6. The final three rows show an actual example, with the individual probabilities, the true cumulative probabilities and values stored in the table.

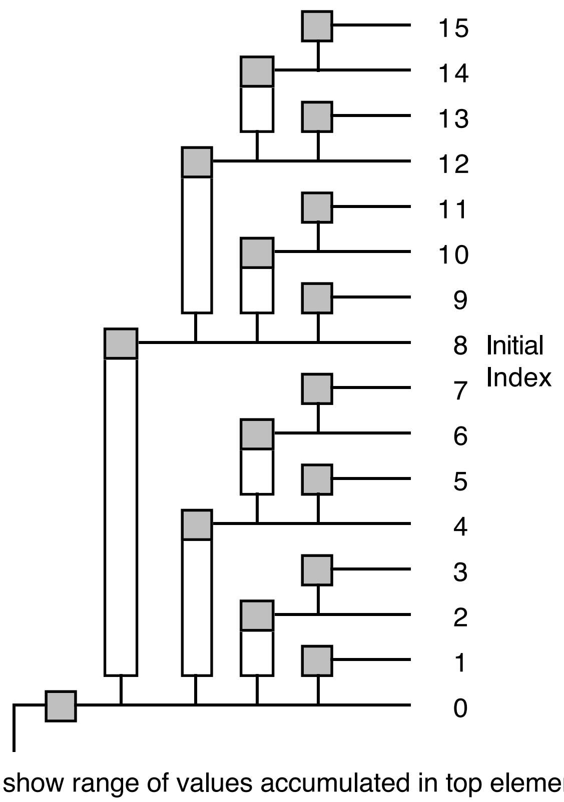


Figure 2. The tree of partial probabilities

To read the cumulative probability for element 11, we first take the value at that index (V[11]). Then, we successively strip off the least-significant 1-bit of the index and add in that element, finally adding V[0]. In this case we add V[10] and V[8] as intermediate values. Referring back to the second row of the table, we see that the sequence V[11]+V[10]+V[8]+V[0] corresponds to the probabilities P[11]+P[9…10]+P[1…8]+P[0], where P[1…8] means P[1]+…+P[8]. The final value is thus P[0…11], which is desired result.

The indexing method generates a tree within the table of partial probabilities, with the structure shown in Figure 2. Each bar represents the range of probabilities held in the array element corresponding to its topmost position (the shaded rectangles). It is clear that traversing the tree from any node to the root will accumulate all of the necessary probabilities.

Alternatively, we can draw the tree in a more conventional form. The table is, in effect, two different trees superimposed on the same table and differentiated by their access algorithms. The "interrogation tree" (to read a cumulative probability) is a decidedly unbalanced tree, as shown in Figure 3. The "update tree" will be described later.

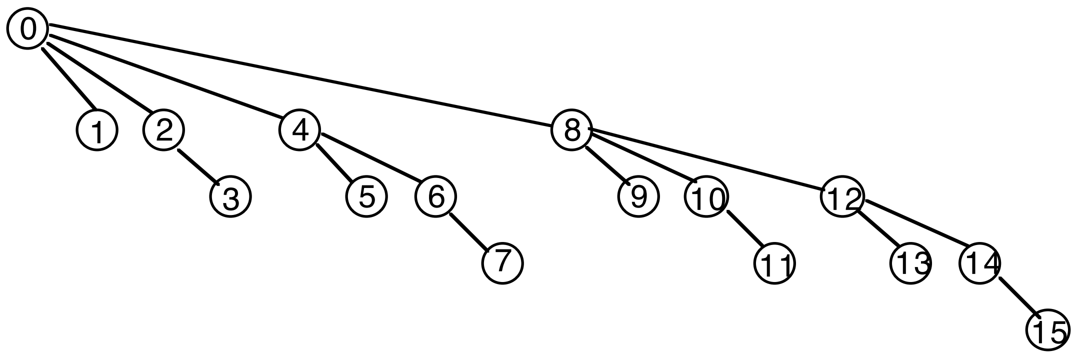


Figure 3. The interrogation tree.

The branching ratio of each node is the number of trailing zeros in its binary representation (each child is formed by converting one of the trailing zeros to a one). The depth at each node is the Hamming weight of its binary index. It is unusual in that its power derives, not from its structure or shape, but from the indexing algorithm. In recognition of the close relationship between the tree traversal algorithms and the binary representation of an element index, the name "Binary Indexed Tree" is proposed for the new structure.

Operations and code to handle the structure.

We need the following functions when processing a symbol in conjunction with arithmetic coding. In all cases the "index" is synonymous with the coding symbol.

1. Read the cumulative probability for an index
2. Update the table according a new probability at a given position.
3. Read the actual probability at a position
4. Find the symbol position within which a given probability lies.
5. Scaling the entire tree by a constant factor (usually halving all counts)

In all of these functions we need efficient ways of isolating and manipulating the least significant 1 bit of a number. Isolating the bit is most easily done (for a 2's complement number) by considering the operation of complementing a positive number. The serial complementing algorithm examines the bits in order from the right (least significant bit), copying all of the least-significant zeros and the rightmost 1 and then complementing each bit to the left. Thus taking the logical AND of a number and its 2's complement isolates the least significant one bit; the bit is present in both values, to its right both values are all-zero, and to its left one or other of the numbers always has a zero. Thus 20 is represented to 8 bits as 00010100 with a 2's complement of 11101100; ANDing the two values gives the result 00000100. With the function BitAnd, as used in many dialects of Pascal, we have that `LSOne:=BitAnd(ix,-ix)`.

From the above discussion we see that the assignment `ix:=ix-BitAnd(ix,-ix)` will strip off the least significant one bit of a binary number. A slightly simpler realisation of the function is `ix:=BitAnd(ix,ix-1)`. The discussion is similar to the above, noting that `(ix-1)` replaces a trailing …10000… by …01111… , leaving unchanged the bits to the left of the rightmost 1. Both of the operations (extracting the bit and removing the bit) are simple and can be done with negligible overhead on most computers.

These techniques assume the most frequent case of 2's complement representation. The case `ix:=BitAnd(ix,ix-1)` works with all representations because it uses only positive numbers which are the same in all cases. A reasonable way to extract the least significant bit is then `ix:=ix-BitAnd(ix,ix-1)` or, for 1's complement, `LSOne:=BitAnd(ix,-ix-1)`. Sign-and-magnitude numbers can be handled by calculating LSOne:=BitAnd(ix,2^k-ix), where 2^k is a power of 2 greater than the table size.

The cumulative probability.

A Pascal function to read the cumulative probability is shown in Figure 4. For this and the following examples, the array Tree contains the appropriate sub-probabilities. The number of iterations is clearly just the number of 1 bits in the desired index.

```
function GetCumul (Ix: integer): integer; { read cumulative value }
    var
       Sum: integer;
    begin
    Sum := Tree[0];                     { initial value }
    while Ix > 0 do
       begin
       Sum := Sum + Tree[Ix];           { include this value }
       Ix := BitAnd(Ix, Ix - 1);        { remove Least Sig one }
       end;
    GetCumul := Sum;
    end;
```

Figure 4. The GetCumul function.

As a simple indication of the cost of reading a value from the table, we can count the number of memory accesses into the data table. For a table of 2^N entries, this is clearly 1+N/2 on average. Note that this is an *average* value only. The combination of an irregular symbol distribution and the non-uniform access costs of the Binary Indexed Tree can lead to considerable variations for real symbol alphabets.

Updating the table

In reading a value we strip off 1 bits and move back towards the start of the table. In updating the table we must increment all sub-probabilities above the position being incremented. Referring to Figure 1, an adjustment to element 9 must be accompanied by adjustments to elements 10 and 12 (those whose ranges cover 9). From 9 we step to 10 (add 1) and then to 12 (add 2). Instead of stripping off the least-significant 1 bit (ie subtracting), we now add it on at each stage to get the next entry to adjust.

```
procedure PutValue (Val, Ix: integer);
   begin
      repeat
      Tree[Ix] := Tree[Ix] + Val;
      Ix := Ix + BitAnd(Ix, -Ix);    { add least-sig one }
      until Ix >= TableSz;
   end;
```

Figure 5. Updating the table.

In the example of Figure 1, if we wish to adjust position 5, the successive indices are 5, then 6 (5+1), and finally 8 (6+2). These three changes affect all of the cumulative probabilities from position 5 up.

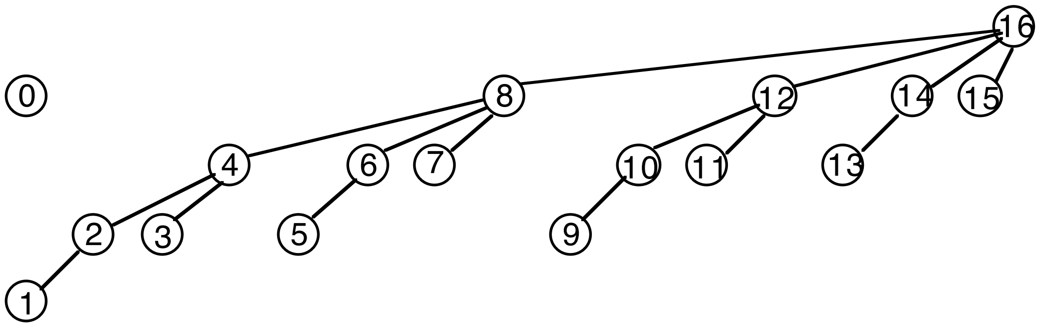


Figure 6. The updating tree.

The tree for updating is essentially the mirror-image of the interrogation tree, with each element resembling its 16-complement in the earlier one. It is shown in Figure 6. Each parent still has a 1 with trailing zeros, but the child indices are formed by successively replacing the first 1, 2, 3 … of those zeros by ones. The interrogation tree has element 0 at its root; this tree has an implicit element 16 at the root and element 0 stands apart as a special case. (Element 16, or its equivalent, is used as the END symbol in Witten's implementation of arithmetic coding and is then a valid part of the table.)

The cost in table references is most easily found by noting that the interrogation and updating trees are mirror-images of each other. The cost of adding a probability into the table is therefore still references to N/2 elements, or N references with separate reads and writes. Once again, we can expect real alphabets to have considerable deviations from this average.

Reading a single probability.

We read a single probability by taking the difference of two adjacent cumulative probabilities. Quite clearly, the two elements will often share a large common path which need not be evaluated as its contributions will cancel out. The method for element “i” is to read the value at node “i”, obtain the parent to node “i” and then trace back from node “i–1” to the parent of node “i”, subtracting off the values traversed. Code is shown in Figure 7.

```
function GetProb (Ix: integer): integer; { read individual probability }
      var
         Val, Parent: integer;
      begin
      Val := Tree[Ix];                      { get the current prob }
      if Ix > 0 then                        { Ix=0 is a special case }
         begin
         Parent := BitAnd(Ix, Ix - 1);     { get the parent node }
         Ix := Ix - 1;                      { the previous node }
         while Parent <> Ix do              { repeat until at prev parent }
            begin
            Val := Val - Tree[Ix];          { adjust for traversed value }
            Ix := BitAnd(Ix, Ix - 1);       { get this parent }
            end;
         end;
      GetProb := Val;
      end;
```

Figure 7. Finding a single probability

The cost is one plus the number of trailing zeros in the index. Half the time (with an odd initial index) it is necessary to read only a single value from the table, for one quarter of the time (indices 2, 6, 10, 12, …) it is necessary to read 2 values, in one of eight cases to read 3 values, and so on. Each term has the form $i \times 2^{-i}$ and the series has a sum to infinity of 2, which value may be taken as a reasonable approximation of the cost in most cases. For a 256 element table the sum is actually 1.93.

Finding an element corresponding to a probability

Note : This method is not satisfactory and has been replaced by that described in Technical Report 110, Jan 1995

The last operation is that of finding the element corresponding to a given cumulative probability. This action is performed by the modified binary search shown in Figure 8.

It is called with the test value and a mask which initially locates the midpoint of the table. (With the 16 element table of the examples here, the initial value of Mask would be 8.) At each stage Index defines the base of the area still to be searched. The midpoint is probed and, if the value is above the midpoint, the value is subtracted off the desired probability and the midpoint becomes the new Index value (or base of the search area). Finally, the Mask value is halved to search at a finer resolution.

```
function getIndex (Prob, Mask: integer): integer;
     var
       Index, TestIx: integer;
     begin
     Index := 0;                      { initial index }
     if Prob > Tree[0] then
       while Mask <> 0 do             { scan all possible bits }
         begin
           TestIx := Index + Mask;    { get trial index }
           if Prob >= Tree[TestIx] then
             begin                    { value in new range }
             Index := TestIx;         { update current index }
             Prob := Prob - Tree[Index]; { revise probability }
             end;
           Mask := Mask div 2;        { scale back test bit }
           end;
     getIndex := Index;
     end;
```

Figure 8. Finding the element, given a probability

The program of Figure 8 fails if the true probability of the element is zero. This is not a problem with the arithmetic coding algorithm of Witten etal which requires non-zero probabilities. There seems to be no efficient programming solution to this problem, but a simple detour is to assume a constant base probability for all values, adjusting the cumulative or real probabilities as they are read.

The average cost in table references is an initial test of the zero element, followed by a probe for each bit of the index and a 50% probability of reading the value to revise the probability (although this last reference may disappear with complier optimisation). The cost in table references, for a 2^N entry table, is then either 1+N, or 1+3N/2. The cost is again logarithmic in the table size.

Scaling the entire tree

Most implementations of adaptive arithmetic coding require that the cumulative probabilities be scaled back as soon as the total probability exceeds some defined threshold. For example, we may halve all probabilities as soon as the total exceeds 16383. Superficially, it appears that as all values are a linear combination of tree entries we can simply halve all of the table entries, but rounding leads to inconsistent entries, with some small probabilities vanishing completely. A simple possibility is to read all of the cumulative probabilities into a work array and then clear and rebuild the tree.

However, it is possible to rebuild the tree in place. First note that when reading values we refer only to entries below the leaf node, while when updating, we modify only those above the leaf node. Therefore, by scanning down the table reading and updating, we will always read only the old, unmodified values. The loop to halve all probabilities just reads the probability for an index and subtracts half that value from the same index. It is shown in Figure 9.

```
for i := TableSz downto 0 do
      addValue(-GetProb(i) div 2,1);
```

Figure 9. Halving all probabilities

Performance and Comparisons

The Binary Indexed Tree technique (BIT) was compared with the MTF algorithm of Witten etal, Jones SPLAY algorithm and Moffat's HEAP algorithm. In all cases the model maintenance involves relatively simple loops which adjust array elements. A simple comparison is the number of accesses to the arrays of the model. While the code in SPLAY and HEAP is more complex than for the other two examples, its quantity and style is more or less in line with the number of memory references. The arithmetic coder itself is taken from Witten etal [5], replacing the model as necessary.

	MTF	SPLAY	HEAP	BIT
bib	32.7	76.3	22.5	17.9
geo	81.3	80.2	25.2	13.8
obj1	83.4	73.1	27.5	13.5
paper1	28.2	69.7	22.1	17.7
paper2	22.6	67.5	21.6	17.7
progc	34.3	71.9	22.6	17.9
progl	24.9	63.0	21.8	17.9
progp	29.2	65.5	22.0	18.1
trans	38.5	70.7	23.2	17.2
skew	5.3	14.6	17.2	18.2

Table 1. Comparative results

Measurements were made on the smaller files of the Calgary Corpus (size about 100 kbytes and smaller) with the results of Table 1. The additional file SKEW contains the pattern "aaaab", repeated to a length of 20,000 bytes.

In all of the realistic cases the Binary Indexed Tree requires fewer data memory references than the other algorithms. All of the textual files have an average cost of about 18 array references for each symbol encoded, compared with an average of about 30 for MTF, 70 for SPLAY and 22 for HEAP. The two binary files are even better, at 13.7 compared with 82 and 26. The critical factor appears to be the actual vocabulary of the file, with the three older methods improving for smaller working alphabets. This effect is particularly marked on the SKEW file, where move-to-front performs particularly well, but the Binary Indexed Tree still behaves as for any text file. Jones, comparing actual execution times for SPLAY and MTF, noted that the splay algorithm was faster only for files with high entropy (such as GEO). Moffat found little real difference in performance between MTF and HEAP. The results here agree with those observations.

The costs given earlier for a 256 entry table predict 8 references to update an element, 5 to read a cumulative probability and 2 to read a single probability, giving a total of 15 references per input byte. As the alphabet actually uses 257 symbols, to allow the End-of-File code, the PutValue routine always refers to the root element (Tree[256]). This adds an extra 2 memory references to each update operation, increasing the previous count to 17. The need to scale (read and update) all values each 18384 input symbols adds another 0.2 references per input symbol. This gives a predicted cost of 17.2 references per byte. The extra 0.5 – 1 references per symbol arise from the non-uniform symbol distribution interacting with the tree structure.

The MTF model uses 3 parallel arrays, with a total of 4 bytes per symbol of the coding alphabet. (We assume here that all integers are 16 bit.) Moffat's HEAP algorithm adds a further integer array, to give a total of 6 bytes per symbol. Jones' SPLAY algorithm uses 4 arrays of integers (although one could be 8 bits), to give 8 bytes per symbol. The Binary Indexed Tree requires only one integer (2 bytes) for each possible symbol and is much more compact than any of the alternatives as well as being faster for most files.

Further applications

When starting arithmetic compression it is sometimes useful to increment counts by more than one to force a faster initial adjustment of the model. One little known problem of the move-to-front algorithm is that it works only for increments by one and that extending it to handle larger increments is not easy without compromising the efficiency. The Binary Indexed Tree does not have this problem.

The simple compact data structure of the Binary Indexed Tree makes it useful for a "brute force" implementation of an Order-1 arithmetic compressor. Briefly, such a compressor uses the known, previous, character to select one of 256 models to encode the next character, leading to a total of 65,536 entries in the model tables. As the move-to-front model needs at least 4 bytes per entry (256K bytes in total), it is usual to combine the models with an LRU structure which retains only important elements. For example, Gutmann [1] requires 10 bytes per element and uses from 3000 elements (for text files) to 20,000 elements for some binaries. The Binary Indexed Tree requires only 2 bytes per element and only 128K bytes to hold the entire model. The storage requirements are therefore reasonable in modern computers and we avoid all of the problems associated with managing the LRU lists.

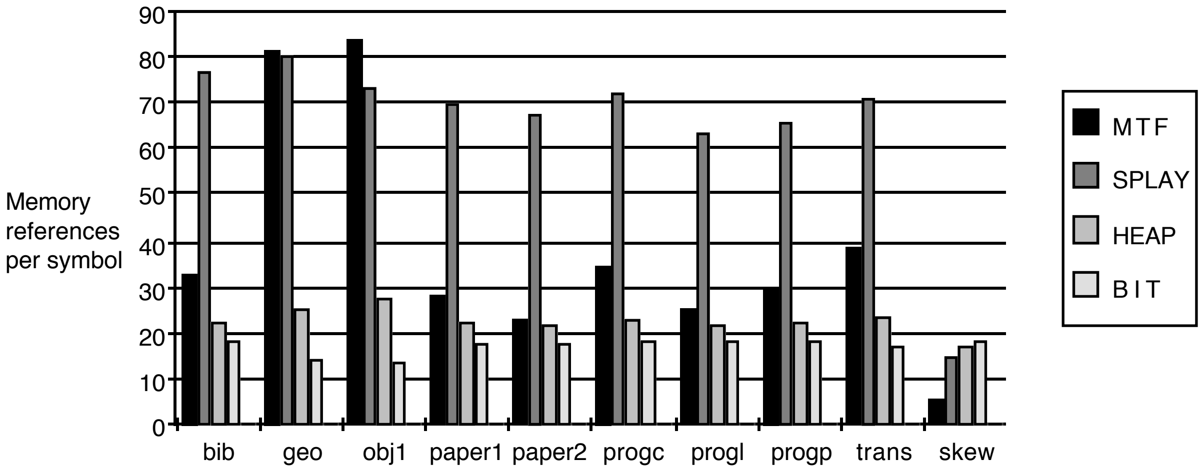
Another potential use for the Binary Indexed Tree is in handling very large alphabets such as occur in word based arithmetic compression [4]. In this case we take advantage of the compact data structure and the logarithmic access cost.

Acknowledgments.

Thanks are due to the University of Auckland for the provision of research facilities, and to Peter Gutmann and Stuart Woolford whose interest provided the incentive for this work.

References.

1. Gutmann, P.C. "Practical Dictionary/Arithmetic Data Compression Synthesis", MSc thesis, University of Auckland, Feb 1992
2. Jones, D.W. "Application of splay trees to data compression", *Comm ACM*, Vol 31 No 8 pp 996–1007 Aug 88
3. Moffat, A. "Linear Time Adaptive coding", *IEEE Trans Info Theory.*, Vol 36, No 2, pp 401–406 Mar 90
4. Moffat, A. "Word-based text compression", *Software–Practice and Experience*, Vol 19, pp 185–198 Feb 89
5. Witten, I.H., Neal, R., and Cleary, J.G. "Arithmetic Compression for data compression", *CACM* Vol 30, No 6, pp 520-540 Jun 87



90
80
70
60
50
40
30
20
10
0
Memory
references
per symbol
bib
geo
obj1
paper1
paper2
progc
progl
progp
trans
skew
MTF
SPLAY
HEAP
BIT