14

Huffman minimum redundancy coding

It has become usual to store data and transmit messages using codes of fixed length such as ASCII. The character set is represented by some number of codewords, each of the same length, which in the case of ASCII is seven binary digits. The result is that it takes the same number of bits to store, or the same bandwith to transmit, all messages with the same number of characters. Of course, if you know in advance that your message is in, say, English, then you know that it is much less likely to contain letter 'z's than letter 'e's. This means that if you use a shorter codeword to represent 'e' than that used for 'z', you can expect to use less store, or bandwidth, for the average message.

In ASCII, the message 'message' is encoded

`1101101`	`1100101`	`1110011`	`1110011`	`1100001`	`1100111`	`1100101`
m	e	s	s	a	g	e

requiring forty-nine bits, whereas by using a code made up of the following representations

$$
\begin{array}{lll}
a & \mapsto & \mathtt{0\ 0\ 0} \\
e & \mapsto & \mathtt{1} \\
g & \mapsto & \mathtt{0\ 0\ 1\ 0} \\
m & \mapsto & \mathtt{0\ 0\ 1\ 1} \\
s & \mapsto & \mathtt{0\ 1}
\end{array}
$$

the same message can be encoded

`0 0 1 1`	`1`	`0 1`	`0 1`	`0 0 0`	`0 0 1 0`	`1`
m	e	s	s	a	g	e

in only seventeen bits. The codewords must be chosen in such a way that none is a prefix of any of the others, to ensure that there can be only one way of decoding a particular coded text.

In a classic paper, published in 1952, David Huffman described an algorithm to find the set of codes that would minimize the expected length of a message, given that the probability of each character's occurring were known. Essentially, his method decides the lengths of codewords, giving the longest to the least likely characters. It then remains to create an arbitrary unambiguous code with codewords of the right lengths.

The terminology of Huffman's paper is a little different from that in use today. He uses the term 'message' to mean an individual character. First of all, the ensemble [= character set], which has size N, is numbered in decreasing order of probability. $L(n)$ is defined to be the (unknown) length of the code for the n^{th} most probable message; it is shown to be required that $i < j \Rightarrow L(i) \leq L(j)$:

> [It is] necessary that the two least probable messages have codes of equal length ... $L(N)$ which are identical except for their last digits. The final digits of these two codes will be one of the two binary digits, 0 and 1. It will be necessary to assign these two message codes to the N^{th} and $(N-1)^{\text{st}}$ messages since at this point it is not known whether or not other codes of length $L(N)$ exist. Once this has been done, these two messages are equivalent to a single composite message. Its code (as yet undetermined) will be the common prefixes of order $L(N) - 1$ of these two messages. Its probability will be the sum of the probabilities of the two messages from which it was created. The ensemble containing this composite message in the place of its two component messages ... contains one less message than the original. Its members should be rearranged if necessary so that the messages are again ordered according to their probabilities. It may be considered exactly as the original ensemble was... .
>
> The procedure is applied again and again until the number of messages in the most recently formed auxiliary ensemble is reduced to two. One of each of the binary digits is assigned to each of these two composite messages. These messages are then combined to form a single composite message with probability unity, and the coding is complete... .
>
> Since each combination of the two messages ... is accompanied by the assigning of a new digit to each, then the total number of digits which should be assigned to each original message is the same as the number of combinations [undergone by] that message.
>
> Having now decided proper lengths of code for each message, the problem of specifying the actual digits remains. Since the combining of messages into their composites is similar to the successive confluences of trickles, rivulets, brooks, and creeks into a final large river, the procedure thus far described might be considered analogous to the placing of signs by a water-borne insect at each of these junctions as he journeys downstream... . The

> code we desire is that one which the insect must remember in order to work his way back upstream.
>
> A method for the construction of minimum-redundancy codes,
> David A. Huffman [19]

Restated more prosaically, the final paragraph identifies the unambiguous set of codewords with a (binary) tree. Each leaf of the tree corresponds to one of the characters. The depth of that leaf, that is its distance from the root, is the length of that character's codeword. The digits of the codeword are the 'address' of the leaf, that is a sequence of instructions for getting to the leaf from the root, say 0 for 'go to the left' and 1 for 'go to the right'.

Representing a coding tree

As usual, the task of representing a data structure in occam amounts to choosing an enumeration for the component parts, so as to map the structure onto an array. The structure in question this time is a binary tree similar to that in Chapter 12, but this tree may be severely imbalanced, and is of unpredictable depth. This means that the simple fixed enumeration, with the children of node i being nodes $2i + 1$ and $2i + 2$, would be unreasonably wasteful of store, so is unsuitable. A better representation, in this case, roots the tree at node zero

```
VAL INT root IS 0 :
```

and uses an array *eldest* to record the index of the leftmost offspring of a node, so that the children of node i are indexed $eldest[i]$ and $eldest[i] + 1$.

Since the root is by definition not the child of any node, the test

$$eldest[node] = root$$

can be used to signify that *node* is a leaf of the tree. In the case of the leaves, it will be necessary to know to which character they correspond. This is most readily recorded in another array of the same size as *eldest* in which the value of $character[node]$ is the character corresponding to that node, if it is a leaf.

The array *eldest* makes it easy to pass 'upstream' from the root of the tree to the leaves. In order to make the 'downstream' journey as efficient, it will be useful to record the inverse of *eldest*, in an array *parent*, such that

$$parent[eldest[node]] = parent[eldest[node] + 1] = node$$

for each non-leaf node, and the inverse of *character* in an array *representative*, which records the index of the leaf corresponding to each character.

It remains to be decided how big these arrays must be. This, of course, depends on the size of the character set being encoded. For the purposes of this example, the (unencoded) character set will be signed, eight-bit significant values, $-128 \leq char < 128$. This allows room for the normal seven-bit characters in the non-negative half range, and room for another, negative, character set which can be used for control information, indicating such things as the end of a message. Since we are admitting negative values, `BYTE` is not the appropriate type for the characters here; we are also going to want to create subscripts into the array *representative* from them, so `INT` is convenient. This is also the natural type to use in the other arrays, as they will store subscripts. In this case, the relevant definitions to make are

```
VAL INT bits.in.character    IS 8 :
VAL INT number.of.characters IS 1 << bits.in.character :
VAL INT number.of.codes      IS number.of.characters :
VAL INT character.mask IS
                   BITNOT ((BITNOT 0) << bits.in.character) :
INT FUNCTION index(VAL INT char) IS char /\ character.mask :
```

The *character.mask* has ones as its *bits.in.character* least significant bits, and zeros elsewhere, and is used by *index* to map signed characters onto non-negative array indices, so that

$$char = character[representative[index(char)]]$$

Now if there are *number.of.codes* leaves in a binary tree, then there will be one less than that number of non-leaf nodes, so the total number of nodes is given by

```
VAL INT size.of.tree IS (2 * number.of.codes) - 1 :
```

and the declarations of the arrays for representing the tree could be

```
[size.of.tree]INT eldest, parent, character :
[number.of.characters]INT representative :
```

As these four arrays constitute the representation of a single data structure, it is convenient to group them together in a single two-dimensional array which can be passed as a complete entity to procedures, and to use abbreviation to give the descriptive names to the component arrays

```
VAL INT number.of.arrays IS 4 :
VAL INT eldest.index IS 0 :
VAL INT parent.index IS 1 :
VAL INT character.index IS 2 :
VAL INT representative.index IS 3 :

[number.of.arrays][size.of.tree]INT tree :
```

```
[]INT eldest IS tree[eldest.index] :
[]INT parent IS tree[parent.index] :
[]INT character IS tree[character.index] :
[]INT representative IS
          [tree[representative.index] FROM
                              0 FOR number.of.characters] :
```

This wastes a certain amount of space, since the *representative* array is not as large as the others, but this is unlikely to be significant.

Constructing a coding tree

Huffman's algorithm proceeds in two stages. First the character set is sorted into descending order of probability of the character's occurrence. Each of the characters will correspond to a leaf of the tree, so you can think of this stage of the process as constructing *number.of.codes* number of leaves. These leaves will be sub-trees of the final coding tree. Since each is just a leaf, they are disjoint, in the sense that they share no nodes with each other, and they are maximal, in the sense that there is not yet any bigger tree of which any is a member.

The second stage of the algorithm repeatedly reduces the size of the collection of maximal disjoint sub-trees, by combining the two lightest trees to make one new composite tree – 'lightest' meaning of least *weight*, where the weight of a leaf is the probability of the corresponding character, and the weight of a larger tree is the sum of the weights of its leaves. Notice that during this second stage, it is guaranteed that any two siblings – children of a common parent – are already adjacent in descending order of weight. This observation, which is taken from Robert Gallager

> A prefix condition code is a code with the property that no codeword is a prefix of any other codeword. A binary tree has the sibling property if each node (except the root) has a sibling, and if the nodes of the tree can be arranged in order of non-increasing probability with each node being adjacent to its sibling. A binary prefix condition code is a Huffman code iff the code tree has the sibling property.
>
> Variations on a Theme by Huffman, Robert G. Gallager [20]

is in fact a non-algorithmic characterization of Huffman codes. It also shows that in the representation chosen for the coding tree, which allocates adjacent elements of the arrays to siblings, it is possible to keep the arrays sorted in descending order of weight. Gallager's proof that this property holds is, essentially, an informal proof of correctness of Huffman's algorithm.

Process 14.1

```
PROC construct.tree([][]INT tree, VAL []REAL32 probability)
  INT left.limit, right.limit :
  [size.of.tree]REAL32 weight :

  PROC construct.leaves([][]INT tree,
                        INT left.limit, right.limit,
                        []REAL32 weight,
                        VAL []REAL32 probability    )
    ... build the leaves of the tree
  :

  PROC construct.other.nodes([][]INT tree,
                             INT left.limit, right.limit,
                             []REAL32 weight             )
    ... join pairs of sub-trees until only one remains
  :

  PROC invert.representation([][]INT tree)
    ... set parent[] and representative[]
  :

  SEQ
    left.limit   := size.of.tree
    right.limit  := size.of.tree

    -- left.limit = size.of.tree
    -- (right.limit - left.limit) = 0

    construct.leaves(tree,
                     left.limit, right.limit, weight,
                     probability                     )

    -- left.limit = number.of.codes
    -- (right.limit - left.limit) = number.of.codes

    construct.other.nodes(tree, left.limit, right.limit, weight)

    -- left.limit = root
    -- (right.limit - left.limit) = 1

    invert.representation(tree)
:
```

Keeping the arrays sorted by weight of node in this way simplifies the finding of the two lightest sub-trees, and if the arrays are filled from the high-index, light, end towards the root, then sub-trees once constructed need not be moved again.

The algorithm is divided into three parts (Process 14.1). Throughout, the collection of maximal disjoint sub-trees consists of those trees rooted at nodes for which

$$\mathit{left.limit} \leq \mathit{node} < \mathit{right.limit}$$

The initialization of the limits makes this collection empty. The process *construct.leaves* introduces a new sub-tree into the collection for each of the characters of the character set, setting its weight according to the probability of the character, maintaining the arrangement of the leaves in descending order, so that

$$\mathit{left.limit} \leq i \leq j < \mathit{size.of.tree} \quad \Rightarrow \quad \mathit{weight}[i] \geq \mathit{weight}[j]$$

The process *construct.other.nodes* combines the two lightest leaves, which are those nearest to *right.limit*. It introduces a new node with the combined weight of these two, adjusting the limits of the collection, and filling in the shape of the tree in *eldest*. Finally, the process *invert.representation* constructs the arrays *parent* and *representative*.

Both *construct.leaves* and *construct.other.nodes* repeatedly create a new node of some given weight, and insert it into the right place between the limits to maintain the weight ordering of the nodes. The determination of this right place and the consequent adjustment of the lighter nodes is done by Process 14.2. Recall that the roots of the collection of maximal disjoint sub-trees of the coding tree constructed so far lie at or to the right of *left.limit* and to the left of *right.limit* and that they are in descending order of weight. This means that the conditional

```
IF
  IF node = left.limit FOR right.limit - left.limit
    weight[node] <= weight.of.new.node
      weight.limit := node
  TRUE
    weight.limit := right.limit
```

sets the *weight.limit* so that

$$\begin{array}{l} \mathit{left.limit} \leq \mathit{node} < \mathit{weight.limit} \;\Rightarrow\; \mathit{weight}[\mathit{node}] > \mathit{weight.of.new.node} \\ \mathit{weight.limit} \leq \mathit{node} < \mathit{right.limit} \;\Rightarrow\; \mathit{weight.of.new.node} \geq \mathit{weight}[\mathit{node}] \end{array}$$

The sequential loop then displaces each of the heavier nodes one place to the left to make room for the *new.node*, and the *left.limit* of the collection is adjusted to compensate. The shapes of the sub-trees remain the same, so that apart from being shifted up by one no changes are necessary to the values in *eldest*. This is because only nodes to the left of the *weight.limit* are moved, but

$$\mathit{node} < \mathit{weight.limit} \quad \Rightarrow \quad \mathit{node} < \mathit{right.limit}$$

Process 14.2

```
PROC insert.new.node([][]INT tree,
                     INT left.limit,
                     VAL INT right.limit,
                     INT new.node,
                     []REAL32 weight,
                     VAL REAL32 weight.of.new.node )
  INT weight.limit :
  SEQ
    IF
      IF node = left.limit FOR right.limit - left.limit
        weight[node] <= weight.of.new.node
          weight.limit := node
      TRUE
        weight.limit := right.limit
    []INT eldest IS tree[eldest.index] :
    []INT character IS tree[character.index] :
    SEQ node = left.limit FOR weight.limit - left.limit
      character[node-1], eldest[node-1], weight[node-1] :=
        character[node],   eldest[node],   weight[node]
    left.limit, new.node := left.limit - 1, weight.limit - 1
    weight[new.node] := weight.of.new.node
:
```

Process 14.3

```
PROC construct.leaves([][]INT tree,
                      INT left.limit, right.limit,
                      []REAL32 weight,
                      VAL []REAL32 probability    )
  VAL INT minimum.character IS -(number.of.characters / 2) :
  SEQ char = minimum.character FOR number.of.characters
    INT new.node :
    SEQ
      insert.new.node(tree, left.limit, right.limit,
                      new.node, weight, probability[index(char)] )
      []INT eldest IS tree[eldest.index] :
      []INT character IS tree[character.index] :
      eldest[new.node], character[new.node] := root, char
:
```

and the construction guarantees

$$(eldest[node] = root) \vee (eldest[node] \geq right.limit)$$

so that no node which is moved is yet a child.

The *insert.new.node* procedure performs the bulk of the work required for the process that creates the leaf nodes (Process 14.3). This inserts a new leaf into the collection, increasing the size of the collection by decreasing the *left.limit*. Process 14.4 then combines the leaves into a tree; it first removes the two lightest sub-trees from the collection, by decreasing *right.limit*, then joins them under a parent whose weight is the sum of their individual weights. Notice that the assignment to *eldest*[*new.node*] maintains the property that there are no children to the left of the *right.limit*. The process is complete when only one tree remains.

Inverting the representation of the tree is a simple task, which involves assigning to *representative* the indices of the leaf nodes, and to *parent* the indices of the nodes that are not leaves, as in Process 14.5.

That completes the code to construct a coding tree from a given probability distribution.

Process 14.4

```
PROC construct.other.nodes([][]INT tree,
                           INT left.limit, right.limit,
                           []REAL32 weight             )
  WHILE (right.limit - left.limit) <> 1
    INT new.node :
    REAL32 new.weight :
    SEQ
      right.limit := right.limit - 2
      new.weight := weight[right.limit] + weight[right.limit + 1]
      insert.new.node(tree, left.limit, right.limit,
                      new.node, weight, new.weight  )
      []INT eldest IS tree[eldest.index] :
      eldest[new.node] := right.limit
:
```

Process 14.5

```
PROC invert.representation([][]INT tree)
  ... abbreviate eldest, parent, character, representative
  SEQ node = root FOR size.of.tree
    IF
      eldest[node] = root
        representative[index(character[node])] := node
      eldest[node] <> root
        SEQ child = eldest[node] FOR 2
          parent[child] := node
:
```

Encoding and decoding using a coding tree

The encoding of any given character *char* is the sequence of 'go left' and 'go right' instructions that Huffman's insect must follow to pass upstream from the root node to the representative node of that character. It is easy enough to construct this code backwards, since floating downstream involved passing from *node* to *parent*[*node*] in succession from the representative node until the root is reached.

Since the result of the encoding should be a sequence of bits, the natural occam 2 type to use is BOOL. In what follows the value TRUE will mean that the insect should turn to the right, and FALSE to the left. As *eldest* records the leftmost offspring of its subscript, *node* is reached by a right turn precisely when it is the younger child of its parent; that is, when it is not *eldest*[*parent*[*node*]]. Therefore the process

```
SEQ
  length := 0
  node   := representative[index(char)]
  WHILE node <> root
    SEQ
      encoding[length] := node <> eldest[parent[node]]
      length           := length + 1
      node             := parent[node]
```

establishes the condition that

$$\bigwedge_{0 \le i < length} node_i = \begin{cases} eldest[node_{i+1}] + 1 & \text{if } encoding[i] \\ eldest[node_{i+1}] & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} node_0 &= representative[index(char)] \\ node_{length} &= root \end{aligned}$$

so that the encoding of *char* can be transmitted in the right order by

```
SEQ i = 1 FOR length
  output ! encoding[length - i]
```

It remains only to decide how much room needs to be allocated to store the *encoding* whilst it is being constructed. Consider the situation when decoding a Huffman encoded character. Before you receive the first bit of the encoding, there are *number.of.codes* possible codes that you might be about to receive. Each bit that you receive divides the set of possible characters into two non-empty subsets: those that are still possible, and those that are now precluded. This means that at most $number.of.codes - 1$ bits will suffice. In fact, in the worst case, this limit is achieved: if each character is more than twice as probable as the next most likely, then the Huffman codes are, in decreasing order of probability

0, 10, 110, 1110, 11110, ...

with encodings that are $number.of.codes - 1$ bits long for each of the two least probable characters. With this bound on the size of the encoding we can write *encode.character* (Process 14.6).

Process 14.6

```
PROTOCOL BIT IS BOOL :

PROC encode.character(CHAN OF BIT output,
                      VAL [][]INT tree,
                      VAL INT char       )
  -- Transmit the encoding of char in tree along output
  ... abbreviate values of eldest, parent, etc. from tree
  VAL INT size.of.encoding IS number.of.codes - 1 :
  [size.of.encoding]BOOL encoding :
  INT length, node :
  SEQ
    length := 0
    node   := representative[index(char)]
    WHILE node <> root
      SEQ
        encoding[length] := node <> eldest[parent[node]]
        length := length + 1
        node   := parent[node]
    SEQ i = 1 FOR length
      output ! encoding[length - i]
:
```

Decoding a stream of bits to determine the character consists of following the 'go left' and 'go right' instructions as they arrive; remembering that TRUE means 'go right' this is achieved by

```
SEQ
  input ? go.right
  IF
    go.right
      node := eldest[node] + 1
    NOT go.right
      node := eldest[node]
```

Enough bits must be accepted to pass upstream, starting from the root node until a leaf is reached. That leaf indicates the decoded character, and so Process 14.7 decodes a single *char*.

Process 14.7

```
PROC decode.character(CHAN OF BIT input,
                      VAL [][]INT tree,
                      INT char         )
  ... abbreviate values of eldest and character from tree
  INT node :
  SEQ
    node := root
    WHILE eldest[node] <> root
      BOOL go.right :
      SEQ
        input ? go.right
        IF
          go.right
            node := eldest[node] + 1
          NOT go.right
            node := eldest[node]
    char := character[node]
:
```

Process 14.8

```
PROC copy(CHAN OF INT source,
          CHAN OF SIGNAL end.of.source,
          CHAN OF INT sink             )
  -- Copy characters from source to sink until a
  -- signal is received from end.of.source
  BOOL more.characters.expected :
  SEQ
    more.characters.expected := TRUE
    WHILE more.characters.expected
      ALT
        INT char :
        source ? char
          sink ! char
        end.of.source ? CASE signal
          more.characters.expected := FALSE
:
```

Having once fixed on a coding tree, the encoding process can be applied repeatedly to a sequence of characters to produce a sequence of bits. This, in turn, by repeated application of the decoding process, can be turned back into the original sequence of characters.

These procedures can be combined in a pair of processes that when executed in parallel duplicate the behaviour of a buffer like Process 14.8, but which transmit the minimum number of bits necessary to communicate the information contained in the sequence of characters.

Assume for the moment that the probabilities of the characters are fixed in advance, perhaps by considering an average over many messages of the type to be sent.

```
VAL [number.of.characters]REAL32 probability IS
                                                    ... :
```

If it is possible to read the message through before sending it, then you can count actual character frequencies, and produce an optimal Huffman code for the message, but of course you will then have to transmit a description of the code with your message!

If one of the character codes is laid aside to indicate the end of a transmission,

```
VAL INT end.of.message IS -1 :
```

then that can be transmitted after the last real character of the message. That means that the end of the message is marked within the sequence of Huffman encodings of the characters.

Process 14.9

```
PROC copy.encoding(CHAN OF INT source,
                   CHAN OF SIGNAL end.of.source,
                   CHAN OF BIT sink             )

  -- Read characters from source, sending their
  -- encodings along sink, until a signal is received
  -- along end.of.source.

  BOOL more.characters.expected :
  [number.of.arrays][size.of.tree]INT tree :
  SEQ
    construct.tree(tree, probability)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      ALT
        INT char :
        source ? char
          encode.character(sink, tree, char)
        end.of.source ? CASE signal
          more.characters.expected := FALSE
    encode.character(sink, tree, end.of.message)
:
```

Process 14.9 translates a stream of characters into a stream of bits representing their Huffman encodings, and marks the end of the stream by

Process 14.10

```
PROC copy.decoding(CHAN OF BIT source,
                   CHAN OF INT sink   )

  -- Read a bit stream from source, decoding it
  -- into characters and send these along sink
  -- until end.of.message is decoded

  BOOL more.characters.expected :
  [number.of.arrays][size.of.tree]INT tree :
  SEQ
    construct.tree(tree, probability)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      INT char :
      SEQ
        decode.character(source, tree, char)
        IF
          char <> end.of.message
            sink ! char
          char = end.of.message
            more.characters.expected := FALSE
:
```

Process 14.11

```
PROC copy.over.serial.medium(CHAN OF INT source,
                             CHAN OF SIGNAL end.of.source,
                             CHAN OF INT sink             )

  -- Copy characters from source to sink until a
  -- signal is received from end.of.source

  CHAN OF BIT serial.medium :
  PAR
    copy.encoding(source, end.of.source, serial.medium)
    copy.decoding(serial.medium, sink)
:
```

sending the encoding of *end.of.message*. Process 14.10 accomplishes the corresponding decoding, terminating when the encoded *end.of.message* value is received and decoded.

These processes can be used at the opposite ends of a serial communications medium where each communication on *serial.medium* is the transmission of a single bit. The process *copy.over.serial.medium* (Process 14.11) is a buffer: provided a sequence of bytes is communicated on *source* which does not include *end.of.message* (if for example they are all seven-bit character codes) either *copy* or *copy.over.serial.medium* can be substituted for the other in any program without affecting the behaviour of the program.

Many communications media are most efficiently used by sending fixed-sized messages, each consisting of a large number of bits. You already have most of the components necessary to implement a process like *copy* which makes efficient use of such a medium. Encoding characters and packing the bits into blocks can be done by executing *copy.encoding* concurrently with the process *pack.bits.into.blocks* (Process 8.20) from Chapter 8 (Process 14.12).

Process 14.12

```
PROC encode.into.blocks(CHAN OF INT source,
                        CHAN OF SIGNAL end.of.source,
                        CHAN OF BLOCK sink          )
  CHAN OF BIT bit.stream :
  CHAN OF SIGNAL end.of.bit.stream :
  PAR
    SEQ
      copy.encoding(source, end.of.source, bit.stream)
      end.of.bit.stream ! signal
    pack.bits.into.blocks(bit.stream, end.of.bit.stream, sink)
:
```

Decoding the characters from the stream of blocks is a slightly trickier task, since the end of the message is determined from the decoded data. The most elegant solution, as seems common in parallel programs, involves a process that throws away unwanted information (Process 14.13). This inputs successively from *source*, ignoring the values that it receives, until a signal is sent to it on *end.of.source*.

Process 14.13

```
PROC discard(CHAN OF BIT source, CHAN OF SIGNAL end.of.source)
  BOOL more.expected :
  SEQ
    more.expected := TRUE
    WHILE more.expected
      ALT
        BOOL bit :
        source ? bit
          SKIP
        end.of.source ? CASE signal
          more.expected := FALSE
:
```

With this, Process 14.14 decodes the bits in a stream of blocks. When *copy.decoding* decodes an *end.of.message* it terminates, causing a signal to be offered for output on *end.of.block.source*, which is a feed-back path to the block unpacking process. At the same time, *discard* absorbs any bits that

Process 14.14

```
PROC decode.from.blocks(CHAN OF BLOCK block.source,
                        CHAN OF INT sink           )
  CHAN OF SIGNAL end.of.block.source, end.of.bit.stream :
  CHAN OF BIT bit.stream :
  PAR
    SEQ
      unpack.bits.from.blocks(block.source,
                              end.of.block.source, bit.stream)
      end.of.bit.stream ! signal              -- 'feed-forward'

    SEQ
      copy.decoding(bit.stream, sink)
      PAR
        end.of.block.source ! signal          -- 'feed-back'
        discard(bit.stream, end.of.bit.stream)
:
```

Process 14.15

```
PROC copy.over.blocked.medium(CHAN OF INT source,
                              CHAN OF SIGNAL end.of.source,
                              CHAN OF INT sink            )
  -- Copy characters from source to sink until a
  -- signal is received from end.of.source
  CHAN OF BLOCK blocked.medium :
  PAR
    encode.into.blocks(source, end.of.source, blocked.medium)
    decode.from.blocks(blocked.medium, sink)
:
```

were left in the last block of the message. When all of the bits of the last block have gone, *unpack.bits.from.blocks* accepts the *end.of.block.source* signal, and terminates causing an *end.of.bit.stream* signal to be sent to terminate the *discard* process.

Process 14.12 and Process 14.14 can be operated concurrently to emulate a buffer (Process 14.15) with each communication on the *blocked.medium* being a transfer of a block of the predetermined fixed size.

The *copy.over.blocked.medium* procedure is not quite as good an implementation of a buffer as is the bit-serial copy. Consider what communications must happen before the first character is output on *sink*. This is a restriction on the contexts in which *copy.over.blocked.medium* can be substituted for *copy*.

Adapting the code to the message

So far, Huffman's assumption that the code is predetermined and remains fixed throughout the transmission of a given message has been accepted. This is reasonable in case the probability distribution of the characters in the message is known in advance, or if the message can be read through in advance. Gallager suggests an alternative encoding that tends in the long run towards the fixed Huffman encoding, but which starts with no knowledge of the probability distribution of the characters, adapting the code as the message is being sent.

Each character is encoded with a Huffman code that would be optimal for a message consisting of all those characters that have gone before it. This encoding technique has the startling property that, since the decoder has already decoded the preceding characters, it can deduce from the received message what code should be used to decode each character. There is no longer a problem in communicating the code as well as the message!

It might seem that Gallager's adaptive coder requires that a new coding tree be constructed for each character of the transmitted and received message. Fortunately, this is not the case: the accumulated character probabilities change little, so the shape of the tree tends to settle down; successive trees are sufficiently similar that it is fairly easy to construct each from its predecessor.

The idea is to write a process *increase.weight*(*tree*, *char*) which modifies the coding tree so as to be consistent with a frequency distribution with one more occurrence of the character *char* than previously. Were relative probabilities used as before, whenever a character arrived the weights of all the components of the tree would have to change, to reflect the reduced likelihood of every other character. This problem is avoided by using absolute character frequencies. The probability is diminished by increasing the implicit denominator, the total number of characters received, while leaving the numerators unchanged except in the branch of the tree containing the representative of the newly arrived character.

To keep track of the accumulated frequencies, the *weight* (now implemented by `[]INT`, rather than `[]REAL32`) must become a permanent part of the representation of the tree

```
VAL INT number.of.arrays IS 5 :
VAL INT weight.index IS 4 :
```

and we will normally refer to this new component by another abbreviation

```
[]INT weight IS tree[weight.index] :
```

The encoding process becomes Process 14.16, and the corresponding decoding process is Process 14.17.

In order to increment the recorded frequency of a character, it is necessary to increment the weight of its representative leaf

```
VAL INT node IS representative[index(char)] :
weight[node] := weight[node] + 1
```

There are two ways in which this may have damaged the structure of the tree. First of all, unless the tree has only the one node, the weight of the parent of *node* is no longer the sum of the weights of its children: it will be necessary to increment the weights of the parent of the node, and all of its ancestors up to the root

```
INT node :
SEQ
  node := representative[index(char)]
  WHILE node <> root
    SEQ
      weight[node] := weight[node] + 1
      node := parent[node]
  weight[root] := weight[root] + 1
```

Secondly, each time the weight of a node, be that the original leaf or one of its ancestors, is increased there is a danger that the ordering of the weights may be upset. If this is the case then it is time to reorganize the tree, and change the encoding.

Assuming that the tree is initially properly ordered, then the ordering will first fail when $weight[node-1] = weight[node]$ and the weight of *node* is about to be incremented. Now, the trees rooted at nodes of equal weight must be disjoint trees, that is either the nodes are siblings, or they have ancestors which are siblings. This follows from the fact that the weight of a node is always less than that of its ancestors, and greater than that of its descendants, so another node with the same weight is neither an ancestor nor a descendant.

To preserve the ordering on the nodes, it would be possible to exchange the trees rooted at *node* and $node - 1$, and then to increment the weight of the light node in its new position. Since there might be many nodes with the same weight, however, this would have to be done repeatedly, shuffling the imminently overweight node leftwards in the tree, by a process like

```
WHILE weight[node-1] = weight[node]
  SEQ
    swap.trees(tree, node, node - 1)
    node := node - 1
```

Process 14.16

```
PROC copy.encoding(CHAN OF INT source,
                   CHAN OF SIGNAL end.of.source,
                   CHAN OF BIT sink             )
  BOOL more.characters.expected :
  [number.of.arrays][size.of.tree]INT tree :
  SEQ
    initialize.tree(tree)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      ALT
        INT char :
        source ? char
          SEQ
            encode.character(sink, tree, char)
            increase.weight(tree, char)
        end.of.source ? CASE signal
          more.characters.expected := FALSE
    encode.character(sink, tree, end.of.message)
:
```

Process 14.17

```
PROC copy.decoding(CHAN OF BIT source,
                   CHAN OF INT sink   )
  BOOL more.characters.expected :
  [number.of.arrays][size.of.tree]INT tree :
  SEQ
    initialize.tree(tree)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      INT char :
      SEQ
        decode.character(source, tree, char)
        IF
          char <> end.of.message
            SEQ
              sink ! char
              increase.weight(tree, char)
          char = end.of.message
            more.characters.expected := FALSE
:
```

An alternative solution is to look for the leftmost node of the given weight, and exchange with that node, directly. The same argument about the weight of a node being less than that of its ancestors shows that there is always a sequence of nodes for which

$$weight[(node - i) - 1] > weight[node - i] = \ldots = weight[node]$$

This leftmost node, indexed $node - i$, is identified, and the exchange performed, by something like

```
IF i = 1 FOR (node - root) - 1
  weight[(node - i) - 1] > weight[node]
    SEQ
      swap.trees(tree, node, node - i)
      node := node - i
```

Unfortunately neither this nor the simpler `WHILE` loop would be legal in occam 2: in the scope of an abbreviation of the *weight* component of *tree*, mention of *tree* is allowed only in making further, non-overlapping abbreviations. Passing *tree* as an argument to *swap.trees* is therefore not allowed, as the procedure call is within the scope of whichever abbreviation makes *weight* accessible in the Boolean condition.

There are several ways to circumvent this clash of scopes. The test could use the unabbreviated forms like *tree*[*weight.index*][*node*]. Another possibility is to make the declaration of *swap.trees* local to the routine that calls it, so that it inherits the abbreviated components of *tree* as free variables. The technique adopted here is to encapsulate the reference to *weight* in a new scope, that of the body of a function *compare.weights* (Process 14.18).

Process 14.18

```
VAL INT lighter IS -1 :
VAL INT same.weight IS 0 :
VAL INT heavier IS 1 :
INT FUNCTION compare.weights(VAL [][]INT tree, VAL INT a, b)
  INT result :
  VALOF
    VAL []INT weight IS tree[weight.index] :
    IF
      weight[a] < weight[b]
        result := lighter
      weight[a] = weight[b]
        result := same.weight
      weight[a] > weight[b]
        result := heavier
    RESULT result
:
```

Combining these fragments yields an algorithm which normally increments the weight of a node, and each of its ancestors; and if this would unbalance the tree, moves the node to where it is safe to increment its weight:

```
INT node :
SEQ
  ... abbreviate representative from tree in next line only
  node := representative[index(char)]
  WHILE node <> root
    CASE compare.weights(tree, node - 1, node)
      heavier
        ... abbreviate parent and weight from tree
        SEQ
          weight[node] := weight[node] + 1
          node := parent[node]
      same.weight
        IF i = 1 FOR (node - root) - 1
          compare.weights(tree, (node-i)-1, node) = heavier
            SEQ
              swap.trees(tree, node, node - i)
              node := node - i
  ... abbreviate weight from tree
  weight[root] := weight[root] + 1
```

The procedure *swap.trees* must exchange the positions in *tree* of two disjoint sub-trees. This must, amongst other things, achieve the effect of

```
eldest[i], eldest[j] := eldest[j], eldest[i]
```

where i and j are the node indices passed as parameters to the call of *swap.trees*. Unfortunately, it is not immediately obvious that $eldest[i]$ and $eldest[j]$ are distinct components of the array *eldest*, so the variables in that particular multiple assignment would not be detectably disjoint.

The programmer's intention that they will always be disjoint can be made explicit by abbreviating the two variables:

```
INT ei IS eldest[i] :
INT ej IS eldest[j] :
ei, ej := ej, ei
```

forcing a compiler to check, or to cause a check during execution, that *ei* and *ej* are distinct. The same effect can be achieved by using the procedure parameter mechanism to make the abbreviations

```
PROC swap.ints(INT a, b)
  a, b := b, a
:
swap.ints(eldest[i], eldest[j])
```

Process 14.19

```
PROC swap.trees([][]INT tree, VAL INT i, j)
  -- Exchange disjoint sub-trees rooted at i and j
  PROC swap.ints(INT a, b)
    a, b := b, a
  :
  ... abbreviate eldest, parent, etc. from tree
  PROC adjust.offspring(VAL INT i)
    -- restore downstream pointers to node i
    IF
      eldest[i] = root
        representative[index(character[i])] := i
      eldest[i] <> root
        SEQ child = eldest[i] FOR 2
          parent[child] := i
  :
  SEQ
    swap.ints(eldest[i], eldest[j])
    swap.ints(character[i], character[j])
    adjust.offspring(i)
    adjust.offspring(j)
:
```

Process 14.19 achieves the exchange of a pair of disjoint sub-trees: first the upstream pointers, *eldest* and *character*, to the nodes are exchanged, then the process *adjust.offspring* restores the downstream pointers that are no longer correct. There is, of course, no need to exchange the weights of the nodes, since they were known to be equal.

The only remaining problem is to decide the shape of the initial coding tree: what encoding should be used to send the first character? The simplest solution would be to construct the initial tree on the assumption that all characters are equally likely to turn up, say $weight[node] = 1$ for all *representative* nodes. This means that, to begin with, the code is a fixed-length one, each character being encoded by *bits.in.character* number of bits.

An alternative technique is to keep in the coding tree only representations of characters that have actually been sent and received. Whenever a character is to be sent for the first time in the message, the code of a special escape 'character' is sent, followed by some standard representation of the new character, say its ASCII code. A new leaf must then be added to the tree to represent the new character.

In order to accommodate the escape character, the space allocated for the tree must be enlarged

```
VAL INT number.of.codes IS number.of.characters + 1 :
```

and, since the tree grows, some way must be found of recording its size. As each escape is the representation of a character that has never occurred at all (you may not yet know which character, but you do know this), it should be given a very low weight. This means that it is reasonable to represent it by the rightmost (least likely) leaf of the tree. Doing this means that a single variable *escape* will serve the purpose of recording which node represents the escape, and which is the rightmost node of the tree. The index of the escape node is an essential part of the coding of the tree, so it is proper to use part of the spare space in the component of *tree* not fully filled by *representative*:

```
INT escape IS
         tree[representative.index][number.of.characters] :
```

Since the value of *escape* changes, it will not do to use it as an initial value for the elements of *representative*. Define, instead,

```
VAL INT not.a.node IS size.of.tree :
```

then creating the initial tree is just a matter of making the escape leaf, and initializing the array of representatives to *not.a.node* with Process 14.20.

Process 14.20

```
PROC initialize.tree([][]INT tree)
  ... abbreviate eldest, escape, etc. from tree
  SEQ
    escape := root
    weight[escape] := 1     -- minimum legal weight
    eldest[escape] := root  -- it is a leaf
    SEQ char = 0 FOR number.of.characters
      representative[char] := not.a.node
:
```

Encoding using the new tree is substantially unchanged, excepting that some provision must be made for sending escaped characters. First of all, the encoding is potentially larger by the *bits.in.character* number of bits in the unencoded representation, so we must define

```
VAL INT size.of.encoding IS
               bits.in.character + (number.of.codes - 1) :
```

The bits of the unencoded character representation can then be stored before the encoding of the escape, to be transmitted after it:

```
SEQ i = 0 FOR bits.in.character
  encoding[i] := ((char >> i) /\ 1) = 1
```

The encoding of the escape changes as the tree is modified. It is of course found by applying to the node representing escape the same algorithm as would be applied to the representative of any genuine character. Process 14.21 thus performs the encoding process. The very first character

Process 14.21

```
PROC encode.character(CHAN OF BIT output,
                      VAL [][]INT tree,
                      VAL INT char       )
  -- Transmit the encoding of char along output
  ... abbreviate values of eldest, parent, etc. from tree
  [size.of.encoding]BOOL encoding :
  INT length, node :
  SEQ
    VAL INT leaf IS representative[index(char)] :
    IF
      leaf <> not.a.node
        length, node := 0, leaf
      leaf = not.a.node
        SEQ
          SEQ i = 0 FOR bits.in.character
            encoding[i] := ((char >> i) /\ 1) = 1
                               -- i'th bit of unencoded char
          length, node := bits.in.character, escape
    WHILE node <> root
      SEQ
        encoding[length] := node <> eldest[parent[node]]
        length, node := length + 1, parent[node]
    SEQ i = 1 FOR length
      output ! encoding[length - i]
:
```

to be sent will be escaped, and since the representative node for *escape* is initially *root* the encoding of the escape will be the null sequence of bits. This means that the first transmitted bit will be the first bit of the unencoded character representation.

Decoding is also similar to decoding with a fixed code, excepting that on receipt of the coding for escape, the bits which made up the unencoded, escaped character must be read and that character reassembled. The first bit of an escaped sequence is `TRUE` precisely when the character is negative, and the decision

```
SEQ
  input ? bit
  IF
    bit
      char := BITNOT 0
    NOT bit
      char := 0
```

therefore correctly extends that bit to the left, so that subsequent bits can simply be shifted in from the right.

Whatever the character to be decoded, escaped or not, the first action must be to accept enough bits to represent a leaf node: initially a sequence of no bits is enough to represent the only leaf, which is *escape*. Then, if the leaf represents a character, that character has been decoded; if the leaf represents the escape, the character code must be reconstructed. Process 14.22 is a modified *decode.character* which does this.

Process 14.22

```
PROC decode.character(CHAN OF BIT input,
                      VAL [][]INT tree,
                      INT char          )
  -- Receive an encoding along input and store the
  -- corresponding character in char
  ... abbreviate values of eldest, character, etc. from tree
  INT node :
  SEQ
    node := root
    WHILE eldest[node] <> root
      BOOL go.right :
      SEQ
        input ? go.right
        IF
          go.right
            node := eldest[node] + 1
          NOT go.right
            node := eldest[node]
    IF
      node < escape
        char := character[node]
      node = escape
        BOOL bit :   -- read bits of signed character code
        SEQ
          input ? bit
          IF
            bit
              char := BITNOT 0
            NOT bit
              char := 0
          SEQ i = 2 FOR bits.in.character - 1
            SEQ
              input ? bit
              char := (char << 1) \/ (INT bit)
:
```

In order to increment the frequency of a character not yet in the tree, it is necessary to be able to construct a new leaf to be the representative of the new character. The new character has never previously been received, so its initial weight should be as low as possible, which means that it belongs next to the *escape* node, at the low-probability end of the tree.

The simplest way of adding a node to the tree is to create a node of zero weight, and then increment its weight (to one) exactly as though it had previously existed. This can be achieved by dividing the escape leaf into three: two new leaves and their parent. The new leaf has no weight when created, so there is no effect on the weights of its ancestors. Notice that a brand new leaf having no weight, the earlier data invariant – that no node has the same weight as its parent – is breached by the *escape* node and its parent. This invariant was used to show that the sub-tree exchanging was correct, so that argument has to be re-constructed. The statement of the invariant must be strengthened: no node, excepting the *escape* node, has the same weight as its parent. This is sufficient (as you should check) because it is never required that the one-node tree rooted at *escape* be exchanged with any other. Process 14.23 is the requisite *create.leaf* process.

Creating a new representative with zero weight makes new characters like those already present in the tree. In either case, recording the arrival of a character is a matter of finding a representative leaf for it, and then incrementing the weight of that leaf and making the consequential adjustments to the tree by Process 14.24.

There is one more refinement that could be added to the process *increase.weight*. At present, once the weight of the *root* (which is the total number of characters sent in the message, plus one for the *escape* node) reaches `MOSTPOS INT`, the next character will cause an arithmetic overflow error. This could be a serious practical problem on machines where the 'natural' integer type is the same as `INT16`, as messages longer than about thirty-two thousand characters would break the transmission medium. Even using explicit `INT64` variables would be setting potential time-bombs for our (many-times-great-) grandchildren, were the encoding and decoding processes built into some long-lived communications system. Using cyclic arithmetic (`PLUS`) generates negative weights, and neither '`>`' nor `AFTER` can be guaranteed to compare the results in the correct sense. Using floating-point numbers would be no solution, either, because the rounding of the result of adding one in order to increment a frequency would eventually leave the number unchanged. The tree would therefore tend to become balanced, representing equal-length codewords.

The proper solution is to detect the problem just before it occurs, and to take avoiding action. It is arguable that the frequency of the earliest characters sent should have little bearing on the encoding of later sections of the message, especially if the data stream is in fact the concatenation of a sequence of different messages separated by some distinguished control value. Two strategies for exploiting this observation suggest themselves: the entire tree could be thrown away, and reinitialized; or less drastically, the existing weights could be scaled down. The disadvantage of the former is that each distinct character used in the rest of the message has to be re-sent in its

Process 14.23

```
PROC create.leaf(INT new.leaf, [][]INT tree, VAL INT char)
  -- Extend the tree by fission of the escape into two new leaves
  ... abbreviate eldest, parent, etc. from tree
  INT new.escape :
  SEQ
    new.leaf, new.escape := escape + 1, escape + 2
    eldest[escape] := new.leaf -- old escape is new parent
    weight[new.leaf],   eldest[new.leaf],   parent[new.leaf] :=
        0,                  root,               escape
    character[new.leaf], representative[index(char)] :=
        char,                new.leaf
    weight[new.escape], eldest[new.escape], parent[new.escape] :=
        1,                  root,               escape
    escape := new.escape
:
```

Process 14.24

```
PROC increase.weight([][]INT tree, VAL INT char)
  INT node :
  SEQ
    ... abbreviate representative from tree for next line only
    node := representative[index(char)]
    IF
      node <> not.a.node
        SKIP
      node = not.a.node
        create.leaf(node, tree, char)
    WHILE node <> root
      CASE compare.weights(tree, node - 1, node)
        heavier
          ... abbreviate parent and weight from tree
          SEQ
            weight[node] := weight[node] + 1
            node := parent[node]
        same.weight
          IF i = [1 FOR (node - root) - 1]
            compare.weights(tree, (node-i)-1, node) = heavier
              SEQ
                swap.trees(tree, node, node - i)
                node := node - i
    ... abbreviate weight from tree
    weight[root] := weight[root] + 1
:
```

escaped form. The latter is not as straightforward as it might seem at first sight, as one has to be careful not to scale anything down to a weight of zero; simply rounding up can lead to breaking the invariants on the tree, requiring the non-leaf nodes to be rebuilt. For the sake of the present example, the former strategy is adopted.

Since the weight of the *root* node is the sum of the (non-negative) weights of its two children, it is clear that it is *weight*[*root*] that is first in danger of overflowing. Thus if that value is tested at the start of the *increase.weight* phase, corrective action can be taken before an error occurs.

Process 14.25

```
PROC increase.weight([][]INT tree, VAL INT char)
  INT node :
  SEQ
    VAL INT limiting.weight IS MOSTPOS INT :
    INT FUNCTION heaviest.weight(VAL [][]INT tree) IS
                                   tree[weight.index][root] :
    IF
      heaviest.weight(tree) < limiting.weight
        SKIP
      heaviest.weight(tree) = limiting.weight
        initialize.tree(tree)

    ... abbreviate representative from tree for next line only
    node := representative[index(char)]
    IF
      node <> not.a.node
        SKIP
      node = not.a.node
        create.leaf(node, tree, char)

    WHILE node <> root
      CASE compare.weights(tree, node - 1, node)
        heavier
          ... abbreviate parent and weight from tree
          SEQ
            weight[node] := weight[node] + 1
            node := parent[node]
        same.weight
          IF i = 1 FOR (node - root) - 1
            compare.weights(tree, (node-i)-1, node) = heavier
              SEQ
                swap.trees(tree, node, node - i)
                node := node - i
    ... abbreviate weight from tree
    weight[root] := weight[root] + 1
:
```

Process 14.25 is the component required for an eternal adaptive coder, provided both ends of the communication share the same natural integer size. If not, some value that both can represent as an `INT` should be substituted for *limiting.weight*.

That completes the adaptive coder. Notice that, since the processes *copy.encoding* and *copy.decoding* have the same interfaces as the corresponding processes in the fixed-code coder, they may be substituted into the example *copy.*— processes. There is no need to change the processes that convey the bit stream from encoder to decoder.