On the Implementation of Minimum Redundancy Prefix Codes

Alistair Moffat and Andrew Turpin

Abstract—**Minimum redundancy coding (also known as Huffman coding) is one of the enduring techniques of data compression. Many efforts have been made to improve the efficiency of minimum redundancy coding, the majority based on the use of improved representations for explicit Huffman trees. In this paper, we examine how minimum redundancy coding can be implemented efficiently by divorcing coding from a code tree, with emphasis on the situation when n is large, perhaps on the order of 10^6. We review techniques for devising minimum redundancy codes, and consider in detail how encoding and decoding should be accomplished. In particular, we describe a modified decoding method that allows improved decoding speed, requiring just a few machine operations per output symbol (rather than for each decoded bit), and uses just a few hundred bytes of memory above and beyond the space required to store an enumeration of the source alphabet.**

Index Terms— **Canonical code, Huffman code, length-limited code, minimum redundancy code, prefix code, text compression.**

I. Introduction

GIVEN a source alphabet $S = [s_1, s_2, \cdots, s_n]$ containing n symbols and an associated set of weights $P = [p_1, p_2, \cdots, p_n]$, a (binary) *minimum redundancy code* $C = [c_1, c_2, \cdots, c_n]$ is an assignment of codewords $c_i \in \{0,1\}^*$ such that c_i is not a prefix of c_j for $i \neq j$ (that is, the set of codewords is *prefix free*), and such that $\Sigma_{i=1}^n \ p_i|c_i|$ is minimized over all prefix-free codes.

Minimum redundancy coding (also known as Huffman coding, after the author of one of the important algorithms for devising minimum redundancy codes [1]) is one of the enduring techniques of data compression [2]. It was used in the venerable PACK compression program, authored by Szymanski in 1978, and remains no less popular today [3], [4].

Many subsequent authors have described improvements to the basic methods for calculating and employing minimum redundancy codes, and these are reviewed in this paper, with emphasis on the situation when n is large, perhaps on the order of 10^6. We then describe a modified decoding method that allows improved decoding throughput, requiring just a few machine operations per output symbol (rather than for each decoded bit), and uses just a few hundred bytes of memory above and beyond the space required to store an enumeration of the source alphabet.

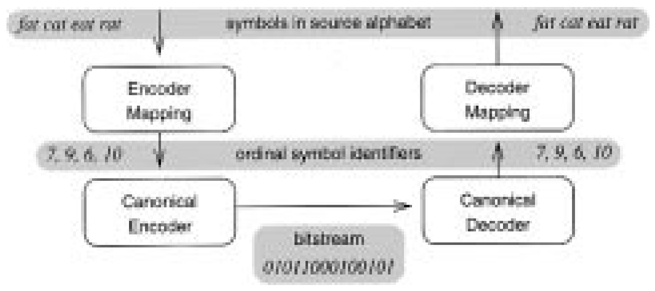

Fig. 1. Structure for minimum redundancy coding.

Our development proceeds as follows. In Section II, we argue that, irrespective of the nature of the source alphabet, minimum redundancy coding can be carried out assuming that S is represented by $[1, 2, \cdots, n]$ and with $p_1 \leq p_2 \leq \cdots \leq p_n$. In Section III, we review methods for calculating codeword lengths. Section IV then shows how those codeword lengths should be used to derive a minimum redundancy code that has the numeric sequence property, and describes a memory-compact method for decoding such canonical codes. An improved method for decoding canonical codes is then presented in Section V, with empirical comparisons with existing decoding methods reported in Section VI. Finally, Section VII concludes our presentation.

II. Mapping the Source Alphabet

One point that appears to be little appreciated in the literature is that there is no disadvantage incurred, and considerable benefit to be gained, from mapping the source alphabet onto integer symbol numbers $1 \cdots n$ such that $p_1 \leq p_2 \leq \cdots \leq p_n$. That is, we believe that both the encoder and the decoder of a practical compression system for some source alphabet carry out two distinct functions. In the encoder, source symbols s_i are first mapped onto ordinal symbol identifiers, and then those symbol identifiers are coded onto an output bitstream. In the decoder, the reverse actions are carried out: first, an ordinal symbol identifier is decoded from the compressed bit stream, and then that identifier is converted, using the inverse mapping, into the correct source symbol s_i. The structure advocated is illustrated in Fig. 1.

Table I lists some data that we will use as a running example throughout this paper. It describes an alphabet of 11 symbols and their weights, perhaps derived from a model representing

Paper approved by E. Ayanoglu, the Editor for Communication Theory and Coding Application of the IEEE Communications Society. Manuscript received July 15, 1996. This work was supported by the Australian Research Council and the Collaborative Information Technology Research Institute. This paper was presented in part at the 1996 IEEE Data Compression Conference, Snowbird, UT, April 1996.

The authors are with the Department of Computer Science, University of Melbourne, Parkville, Victoria 3052, Australia.

Publisher Item Identifier S 0090-6778(97)07276-0.

TABLE I
EXAMPLE ALPHABET AND CORRESPONDING WEIGHTS

i	1	2	3	4	5	6	7	8	9	10	11
s_i	*bat*	*cat*	*cat*	*fat*	*hat*	*mat*	*oat*	*pat*	*rat*	*sat*	*vat*
p_i	8	21	8	9	23	3	10	7	21	5	6

TABLE II
MAPPING TABLES: (a) FOR ENCODING, (b) FOR DECODING

s_i	*bat*	*cat*	*cat*	*fat*	*hat*	*mat*	*oat*	*pat*	*rat*	*sat*	*vat*
r_i	5	9	6	7	11	1	8	4	10	2	3

(a)

r_i	1	2	3	4	5	6	7	8	9	10	11
s_i	*mat*	*sat*	*vat*	*pat*	*bat*	*eat*	*fat*	*oat*	*cat*	*rat*	*hat*

(b)

some simple book. Note that the alphabet has quite deliberately been chosen to be words rather than single letters; the use of single letters implies a dense alphabet that can be used to index a table, whereas in general, the alphabet will be sparse in some universe, and direct indexing impossible.

Table II shows the encoding and decoding mappings used to convert the symbols into ordinal identifiers in increasing probability order, and the mapping used by the decoder to convert ordinal identifiers back to symbols. The mapping r is such that if $r_i < r_j$ for some i and j, then $p_i \leq p_j$. The encoding mapping can be stored in any convenient form: as a sorted list of symbol identifiers as is implicitly suggested in Table II(a), as a hash table, as a binary search tree, or using a minimal perfect hash function. The exact structure used is immaterial to the discussion we pursue here, the point to note being that, during encoding, it allows compression tokens to be converted to ordinal symbol identifiers. The decoding mapping is simpler, as the index is an ordinal value in the range $1 \cdots n$ which allows the mapping to be stored as an array indexed by an ordinal identifier.

For example, the stream *fat cat eat rat* would be represented as "7, 9, 6, 10" by the encoder and transmitted as such; and then, upon receipt of "7, 9, 6, 10," the decoder would reconstruct *fat cat eat rat* through the use of the reverse mapping.

Use of such a mapping does not expand the memory requirements of either the encoder or the decoder. During decoding, an enumeration of the source alphabet must be stored for output, and storing the source alphabet S sorted using r_i as a key rather than using i as the key costs no extra memory. In the encoder, an efficient dictionary data structure using s_i as the search key needs to be employed to locate symbols quickly, and storing r_i in this structure costs no more than storing c_i, the actual codeword. If c_i is stored, we have a codebook in the conventional sense, and storing r_i instead yields, as we shall see below, the same functionality, but with considerably more versatility.

The stored (as compared with the in-memory) representation of the codebook can be even more compact. All that needs to be recorded against each symbol s_i is the length $l_i = |c_i|$ of the corresponding codeword since r_i can be inferred from an approximate sorting process based upon l_i rather than the exact sorting process based upon p_i. That is, once l_i has been calculated using one of the methods described below, the mappings might be reassigned based upon decreasing l_i values rather than increasing p_i values, with the symbol identifiers s_i used as a secondary key. If the maximum codeword length $L = \max_{1 \leq i \leq n} l_i$ is then determined, the cost of storing the r_i values can be $n\lceil \log_2 L \rceil$ or fewer bits. Since L is typically very much less than n, this represents a significant saving over the $n \log n$ bits required to store r_i values explicitly. The actual r_i values must, of course, be recreated from the l_i values before either the encoder or decoder can operate in an efficient manner. Hankamer [5] considers in detail methods for transmitting the codebook component of a minimum redundancy representation for some stream of data.

III. CALCULATING CODEWORD LENGTHS

Given the use of the encoder and decoder mappings, we may now suppose (primarily for notational convenience) that the source alphabet is given by $S = [1, 2, \cdots, n]$.

Before coding can proceed, codeword lengths l_i must be calculated. There is no need at this stage to finalize actual codewords c_i as the compression will be the same no matter what prefix-free bit patterns are assigned as codewords, provided that they are of the same length as those in a minimum redundancy prefix code. Below, we shall describe a mechanism for assigning a set of codewords that allows a particularly elegant encoding and decoding regime to be employed.

Huffman's famous algorithm [1] can be employed to calculate codeword lengths, and takes $O(n \log n)$ time and about $5n$ words of memory to process an n symbol alphabet, not assuming any particular ordering of the symbols. In this method, an explicit code tree is constructed and codewords c_i are read off by labeling each edge in the tree with either a "0" or a "1." That is, l_i is simply the depth in the Huffman tree of the leaf corresponding to s_i with $p_1 \leq p_2 \leq \cdots \leq p_n$.

There are also more efficient methods. Based upon an observation due to Van Leeuwen [6], Katajainen and Moffat [7] have recently described an economical implementation of Huffman's method that, given as input a probability-sorted list of symbol frequencies, generates *in situ* the lengths of the corresponding minimum redundancy codewords. This in-place method requires $O(n)$ time and $O(1)$ additional space, and operates quickly in practice. Compared to the use of Huffman's method, which does not require sorted probabilities, the Katajainen and Moffat technique does require the preapplication of a sorting algorithm, which takes $O(n \log n)$ time and so dominates the code calculation phase. Nevertheless, the use of a method like Quicksort to perform the ordering means that the complete calculation, including sorting, can be done faster than through Huffman's algorithm, which must be supported by a priority queue data structure such as a heap; this is an additional benefit of the probability-sorted mapping espoused above. Other efficient methods for calculating code-

TABLE III
EXAMPLE ALPHABET AND CORRESPONDING CODEWORD LENGTHS

i	1	2	3	4	5	6	7	8	9	10	11
p_i	3	5	6	7	8	8	9	10	21	21	23
l_i	5	5	4	4	4	4	4	3	3	3	2

TABLE IV
CODEWORDS FOR EXAMPLE ALPHABET

i	1	2	3	4	5	6	7	8	9	10	11
l_i	5	5	4	4	4	4	4	3	3	3	2
c_i	00000	00001	0001	0010	0011	0100	0101	011	100	101	11

word lengths on probability-sorted alphabets are also available [8], and under certain circumstances, can operate extremely quickly indeed. Continuing with the example, Table III shows the codeword lengths that would be generated for the alphabet of Table II; the table ordering is now that of the encoder mapping table, and l_i is the length (in bits) of the codeword assigned to the ith least frequent source symbol.

IV. CANONICAL CODING

The litmus test of the "usefulness" of a set of codeword lengths is the Kraft inequality. If a set of lengths l_i is such that $K = \Sigma_{i=1}^{n} 2^{-l_i}$ is less than or equal to 1, then it is possible to assign a set of codewords that has the prefix-free property. To see this, suppose that l_i is the given list of codeword lengths, L is the length of a longest codeword, and that m_ℓ is the number of codewords of length ℓ, for $1 \leq \ell \leq L$. Then the $j+1$st of the m_ℓ codewords of length ℓ should be the ℓ-bit binary integer $j + base[\ell]$, where

$$base[\ell] = \left\lceil \frac{\left(\sum_{k=\ell+1}^{L} m_k \cdot 2^{L-k}\right)}{2^{L-\ell}} \right\rceil \quad (1)$$

When $K = 1$ (as is the case for minimum redundancy codes) and the codeword lengths are nonincreasing (which is always possible if the symbol probabilities are nondecreasing), the ith codeword c_i can be calculated as the l_i low-order bits of $(\Sigma_{j=1}^{i-1} 2^{L-l_j})/2^{L-l_i}$. Table IV lists the codewords assigned for the example alphabet using this technique.

Observe the regular pattern—all codewords of a given length are consecutive binary integers. This is known as the *numerical sequence property*, and the full arrangement is a *canonical code* [9]–[11]. The key advantage of this regular organization of codewords is that it allows fast encoding and decoding using just two L-word lookup tables, one for the array *base* described by (1), and a second *offset* array that records the smallest ordinal symbol number (that is, r_i value) for each distinct codeword length. In particular, it is not necessary for either the encoder or decoder to maintain a codebook, nor any form of code tree. If the input alphabet was not probability-sorted, then it would be necessary to fully enumerate the codewords; hence, the claim above that the encoder and decoder mapping tables are "free." Note also that, in general, as it is for the example of Table IV, the set of codewords so obtained could not, in fact, be achieved by a slavish labeling of edges in the tree constructed by Huffman's algorithm, and so in this restrictive sense, the methodology we are advocating here uses codes that are no longer "Huffman," even though they are minimum redundancy. Huffman codes are a proper subset of the set of codes that are minimum redundancy, and in practice, we are interested in selecting a member of the larger set.

TABLE V
ARRAYS *base* AND *offset* FOR CANONICAL CODING

ℓ	1	2	3	4	5
base[ℓ]	2	3	3	1	0
offset[ℓ]	12	11	8	3	1

The *base* and *offset* arrays for the example alphabet are shown in Table V. For example, the integer value of the first 3-bit codeword (that is, *base*[3]) is 3, and the symbol corresponding to that codeword (that is, *offset*[3]) is the eighth in the (sorted) alphabet. Given these two arrays, encoding a *symbol_id* is straightforward, and is shown as Algorithm CANONICAL-ENCODE.

```
Algorithm CANONICAL-ENCODE
1) Determine the least length such that offset[length] $\leq$ symbol_id.
2) Set code $\leftarrow$ base[length] + (symbol_id − offset[length]).
3) Output the length least significant bits of code.
```

For example, to code the token *fat*, the mapping of Table II(a) is used to convert to a *symbol_id* of 7, which is coded with *length* = 4, and thus *code* = 1 + (7 − 3) = 5, yielding a resultant codeword of 0101. Step 1 of Algorithm CANONICAL-ENCODE can be accomplished by either a linear search or, if memory space is not of concern, via an index that directly maps each *symbol_id* to the corresponding codeword length. Note that use of a linear search is not asymptotically expensive, so the speedup achieved by the use of a table of l_i values is relatively small. Indeed, since the number of iterations of a linear search is exactly the number of bits output, the total cost of encoding (assuming a standard random access machine model of computation) is $O(b+m \cdot f(n))$ for both linear search and table lookup, where b is the number of bits produced, m is the number of symbols encoded, and $f(n)$ is the cost of looking up a symbol s_i in the encoder mapping for an alphabet of n symbols to determine the corresponding r_i value. If a balanced tree dictionary structure is used to represent the source alphabet and symbols can be compared in $O(1)$ time, then $f(n) = O(\log n)$, and if s_i can be used to directly index an array of r_i values (if the source alphabet S is dense integers over some known range or if a perfect hash function is employed), then $f(n) = O(1)$.

Decoding a symbol from a stream of bits is only slightly more difficult: it involves a linear search through the array *base*, adding one bit at a time to a codeword *code*, and then accessing the corresponding symbol via a direct index into the decoder mapping. The time taken is thus $O(m+b)$, where m is the number of symbols processed, b is the number of bits in the compressed bit stream, and it is assumed that it takes $O(1)$ time to output a symbol.

```
Algorithm CANONICAL-DECODE
  1) Set code ← the next bit from the input stream, and
     set length ← 1.
  2) While code < base[length] do
     a) Set code ← LeftShift(code, 1).
     b) Insert the next bit from the input stream into the
        least significant bit position of code.
     c) Set length ← length + 1.
  3) Set
     $symbol\_id \leftarrow offset[length] + (code - base[length])$.
```

For example, if the incoming bit stream is "01011000100101," *code* takes the values 0, 1, 2, and 5, respectively, as *length* is incremented through 1, 2, 3, and 4; and the *symbol_id* of $3+(5-1)=7$ is generated by step 3, indicating that the codeword 0101 represents *fat*, as required. The decoder then restarts, and examines the remaining bit stream "1000100101" in the same incremental manner.

The tight loops and highly localized memory reference pattern of these processes mean that both CANONICAL-ENCODE and CANONICAL-DECODE execute quickly; and the need for just two L-word arrays means that canonical codes are ideally suited to large alphabets [4]. "Huffman" decoding as a process involving the bit-by-bit traversal of an explicit code tree is a convenient simplification when first describing the method to students, but should not be employed in practical compression applications, despite the advice contained in a number of recent papers. For example, Hashemian [12] articulates a "tree clustering algorithm to avoid high sparsity of the tree," and uses a canonical arrangement of codewords to allow a more compact storage of an explicit tree when, in fact, there is no need to store a tree at all. Similarly, McIntyre and Wolff [13] describe "an extremely efficient storage implementation of the Huffman tree shape," and Bassiouni and Mukherjee [14] develop an approach for decoding k-bit trees in an effort to speed up standard Huffman tree coding. Tanaka [15] represents the tree structure by a two-dimensional array which can be applied for the decoding of Huffman codes, again unnecessarily. The only situation when an explicit decode tree is required is when canonical codes cannot be used. One such special case is alphabetic minimum redundancy coding, when a probability-sorted alphabet may not be assumed since the ordering of the codewords for the symbols is important.

As described in CANONICAL-DECODE, the decoding process does, however, manipulate individual bits, and this bit shifting occupies a nontrivial fraction of the decoding time. A number of authors have considered this problem, and have devised interesting mechanisms for performing fast decoding without individual bit manipulations. One such technique is due to Choueka *et al.* [16] (see also [17] for an apparently independent description of some of the same ideas). In this method, a finite-state machine is constructed that allows the input to be processed in units of k bits, with different lookup tables employed for each possible partial-code prefix unresolved from the previous k-bit unit. The drawback of this method is the memory space required since, for an alphabet of n symbols and k-bit decoding tokens, the space required might be as large as $O(n2^k)$. In the application considered below, $n \approx 290\,000$ and $k=8$ or $k=16$ would be reasonable choices, and the resultant need for a gigabyte or more of memory is daunting. Choueka *et al.* reduced the memory space by allowing one symbol per token to be decoded in a partially bit-by-bit manner, but the space required is still at least $O(n \log n)$. Moreover, with a large alphabet, it is highly likely that most symbols will span more than one 8-bit byte, the most natural choice of token, and so this extension is also of limited practicality. For word-based compression, for example, the average code length is typically 10–12 bits. Another advantage of the canonical method as described here is that it sits well with typical current cache-based computer architectures. The encoding and decoding loops span just a few instructions, and for the most part, the data access pattern is also highly localized, so fast loop execution is the result.

In the next section, we describe an alternative mechanism that still performs bit-shifting operations, but on a per-symbol basis rather than a per-bit basis. The result is improved decompression throughput with little or no extra decode-time memory requirements.

V. Fast Decoding

Suppose that an array *lj_base* is initialized to contain "left-justified" bit strings rather than the right-justified values assumed above for the array *base*. That is, suppose that *lj_base*[*i*] = *LeftShift*(*base*[*i*], $w-i$), where w is some convenient integer not smaller than the length L of a longest codeword. Suppose also that a value V is maintained as a w-bit window into the compressed bit stream (we note that a similar left-justified representation was used in the proposal of Choueka *et al.* [16], and that the buffer V is also reminiscent of the mechanism used during the decoding of arithmetic codes [18]). Then the previous canonical decoding process described in Algorithm CANONICAL-DECODE can be rewritten as follows.

```
Algorithm ONE-SHIFT
  1) Determine the least length such that lj_base[length]
     ≤ $V$.
  2) Set symbol_id ← offset[length]
     + RightShift($V$ − lj_base[length], $w$ − length).
  3) Set $V$ ← LeftShift($V$, length), and insert the next
     length bits from the input stream into the now vacant
     length least significant bit positions of $V$.
  4) Return symbol_id.
```

Because the length of the next codeword is calculated before any of the bits comprising the codeword are consumed, each codeword can be extracted with just one pair of shift operations, a considerable reduction in processing effort on architectures that employ barrel shifters rather than serial shifters.

TABLE VI
ARRAY *lj_base*, ASSUMING $w = 5$

ℓ	1	2	3	4	5
$lj\_base[\ell]$ (binary)	100000	11000	01100	00010	00000
$lj\_base[\ell]$ (decimal)	32	24	12	2	0

TABLE VII
ARRAY *start*, ASSUMING $x = 2$ AND $x = 3$

V_x	0	1	2	3	4	5	6	7
$start[V_x]$, $x = 2$	4	3	3	2				
$start[V_x]$, $x = 3$	4	4	4	3	3	3	2	2

Table VI shows the values of *lj_base* stored for the example alphabet, first as bit strings and then as decimal integers. In the table, it is assumed that $w = 5$. In practice, w would be chosen either as the next multiple of eight not less than L or as the machine word size. Note that some initial entries in *lj_base* might be 2^w and require $w + 1$ bits to represent, a problem if w is set to the word size for the architecture being used, but that this need can be circumvented by altering the first step of the algorithm so that it only considers *lengths* that are legitimate code lengths.

Further savings are possible in step 1 of this process. By incrementing *length* one bit at a time, Algorithm ONE-SHIFT is, in effect, performing a linear search through the array *base*, an array which, in fact, contains a sorted list of integers, and thus allows the use of binary search instead. For typical large alphabets (with an average codeword length of, say, 10 bits) and $L = 32$, binary search might halve the average number of values of *lj_base* examined, at the cost of a more complex looping structure.

However, binary search is no less "blind" than linear search, and supposes that each entry in *lj_base* is equally likely to be the target of the search. In fact, the exact frequency distribution of access to the various entries in *lj_base* is known once the code that it serves has been constructed, and so an optimal search structure based on that distribution can be calculated, and should further reduce the number of inspections of *lj_base* values. To implement this approach, the encoder must first determine the code lengths for each symbol and calculate the total number (over all symbols encoded) of codewords emitted of length ℓ, for $1 \leq \ell \leq L$. A minimum redundancy alphabetic prefix code (or optimal binary search tree) can then be constructed for the distribution given by these access frequencies. Finally, tables describing the optimal search can either be stored explicitly with the encoded output, or the encoder can directly generate program code that, when compiled, forms the kernel of the decoder. In the latter case, the decoder is not fully instantiated, and cannot be compiled until after the encoder is executed; this is the approach we adopted in our experiments. To illustrate the way these hard-coded *lj_base* values are embedded into the searching phase of Algorithm ONE-SHIFT, Fig. 2 shows part of the program generated for the test data described in the next section, using $w = 32$. The constants in the if statements are, of course, the *lj_base* values, and the sequence of cascading tests controls the flow of the search. Note how *length* $= 8$ is determined after just two comparisons, while *length* $= 26$ requires nine accesses to the in-line *lj_base* values. Binary search for this particular data set would require either four or five *lj_base* accesses, and linear search between 1 and 21 (assuming that the linear search starts at *lj_base*[5], where 5 is the shortest codeword length). Linear and binary search strategies can also be partially evaluated in this way; and, for example, the linear-search decoder for the same test data results in a nested set of 21 cascading if statements.

While elegant from an algorithmic point of view, the use of an optimal search means that the decoding process must either be table driven with an extra level of indirection at each comparison or, as in our experiments, "hard coded" for a particular probability distribution. An alternative method is to retain the linear search, but use a short prefix of V to indicate an initial value of *length* at which the search should commence. Suppose that V_x is the integer value of the x most significant bits of V, that is, $V_x = \mathit{RightShift}(V, w - x)$, and that *start* is a table of 2^x entries, with $start[V_x]$ recording the smallest value ℓ such that either V_x is a legitimate prefix of an ℓ-bit codeword, or there is an ℓ-bit codeword that is a prefix of V_x. That is, $start[V_x]$ is the least value ℓ for which $V_x \geq \mathit{RightShift}(lj\_base[\ell], w - x)$. Table VII lists values for the array *start* for both $x = 2$ and $x = 3$ for the example alphabet.

Given a precalculated *start* array for some value x, the first step of the decoding process can then be accomplished by the following.

```
Algorithm TABLE-LOOKUP
1) (Replacing step 1 of Algorithm ONE-SHIFT):
    a) Set $V_x \leftarrow \mathit{RightShift}(V, w - x)$.
    b) Set $length \leftarrow start[V_x]$.
    c) If $length > x$ then
           While $lj\_base[length] > V$ do
               Set $length \leftarrow length + 1$.
```

Note the if statement at step 1c. Codewords of x or fewer bits must give rise to V_x values that uniquely determine the codeword length, and this is exploited to minimize the effort involved in searching the *lj_base* table. Indeed, the test used in Algorithm TABLE-LOOKUP is somewhat pessimistic since there may be further entries in the *start* array that also exactly determine the corresponding codeword length. For example, with $x = 2$, the entry for $V_x = 2$ determines correctly that *length* must be 3, and when $x = 3$, the entry for $V_x = 1$ is final.

As an example, consider again the example bit stream "01011000100101." If $x = 2$, then $V_x = 01_2$ and $start[V_x] = 3$. The while loop is entered, but only one increment on *length* (and two *lj_base* comparisons) is required (because $lj\_base[3] = 12_{10}$ is greater than $V = 01011_2 = 11_{10}$) before the correct value of *length* $= 4$ is determined. If $x = 3$, then $V_x = 010_2$, and *length* is correctly initialized to 4 at step 1b. From this example, it can be seen that if x is moderately large, the linear search from $start[V_x]$ will usually terminate having

```
if (V < 3003121664)
    if (V < 1167589376)
        if (V < 614858752)
            if (V < 289472512)
                if (V < 129843200)
                    if (V < 57090048)
                        if (V < 23427072)
                            if (V < 13480448)
                                if (V < 6151552)
                                    length = 26;
                                else
                                    length = 25;
                            else
                                length = 24;
                        else
                            if (V < 37664768)
                                length = 23;
                            else
                                length = 22;
                    else
                        if (V < 86544384)
                            length = 21;
                        else
                            length = 20;
                else
                    if (V < 195592192)
                        length = 19;
                    else
                        length = 18;
            else
                if (V < 421265408)
                    length = 17;
                else
                    length = 16;
        else
            if (V < 844890112)
                length = 15;
            else
                length = 14;
    else
        if (V < 1948254208)
            if (V < 1529872384)
                length = 13;
            else
                length = 12;
        else
            if (V < 2353004544)
                length = 11;
            else
                if (V < 2625634304)
                    length = 10;
                else
                    length = 9;
else
    if (V < 3456106496)
        length = 8;
    else
        if (V < 3892314112)
            if (V < 3690987520)
                length = 7;
            else
                length = 6;
        else
            length = 5;
```

Fig. 2. Optimal search strategy for WSJ words.

examined only a small number of values of *lj_base*. Moreover, it is the symbols with short codes—that is, the most frequent symbols—for which *start* provides the most accurate length indication; in many cases, the if statement at step 1c will avoid the need for any examination of *lj_base* at all.

If x is chosen to be L, the length of the longest codeword, then *start* becomes an array of 2^L entries that deterministically records the length of the next codeword stored in V. This technique for decoding is apparently part of the "folklore" of computing; for example, Hashemian [12] supposes, as his starting point, that the decoder for a code of maximum length L bits will require 2^L words of memory, and then seeks to reduce the space. While fast (no searching of any kind is involved), the use of a table of size 2^L is potentially extremely wasteful of memory space. For example, if there is a 1-bit codeword in the code, then half of the table entries will contain the same value. Hashemian [12] reduces the memory space by using k bits at a time to index a set of tables each of 2^k entries; each entry then indicates either that a codeword has been completed, or the address of a subsidiary table that should be indexed using the next k bits of the input stream. Compared to the 2^L-entry approach this k-bit at a time technique reduces the storage space at the expense of slower decoding speed. In suggesting here that $x < L$ and that a single table be used, we anticipate the best of both of these worlds—on average, only a small amount of searching is incurred, so decoding is still very fast; yet only a small amount of auxiliary memory is required. Below, we give experimental results that show that use of $x = 8$ (that is, a *start* array of 256 entries) gives fast decompression with an alphabet for which $n \approx 290\,000$ and $L = 26$.

VI. Empirical Evaluation

To test the methods described here, we embedded them into existing software for undertaking word-based zero-order compression on large document collections [4]. One typical collection we have been working with is WSJ, about 510 Mbytes of English text drawn from several years of the *Wall Street Journal*, part of the large *TREC* corpus [19]. The word-based model emits codes for words and nonwords in a strictly alternating manner, and so gives access to two distinct sets of symbols: an alphabet of words, which is large $(n \approx 290\,000)$ and has a high self-entropy (11.2 bits/symbol and $L = 26$); and an alphabet of nonwords, which has a smaller alphabet $(n \approx 8\,900)$ and a low self-entropy (2.5 bits/symbol, again with $L = 26$). In each case, there are approximately $m = 87\,000\,000$ symbols to be coded. We thus ran three experiments—coding only the words of WSJ, coding only the nonwords, and coding both words and nonwords to achieve full lossless compression.

In terms of compression effectiveness, the full lossless word-based model reduces WSJ to 28.5% of the original size, including the lexicons of words and nonwords. As a comparative benchmark, the well-known Gzip compression program [3], when applied to the same collection, compresses it to 36.8%, and in the same test harness, decodes at a rate of 110.6 Mbytes/min. The Gzip program uses canonical codes

TABLE VIII
DECODING RATE ON WSJ, MBYTES/MIN

Method	Words	Non-words	Both
CANONICAL-DECODE	74.9	62.3	76.7
ONE-SHIFT, linear search	102.8	64.9	95.2
ONE-SHIFT, binary search	102.4	46.8	85.9
ONE-SHIFT, optimal search	118.6	65.2	106.3
TABLE-LOOKUP, $x = 8$	124.8	62.6	111.4

TABLE IX
AVERAGE NUMBER OF ACCESSES TO *base* OR *lj_base* PER SYMBOL

Method	Words	Non-words	Both
CANONICAL-DECODE	11.22	2.46	6.91
ONE-SHIFT, linear search	11.22	2.46	6.91
ONE-SHIFT, binary search	4.79	4.95	4.87
ONE-SHIFT, optimal search	4.12	1.95	3.03
TABLE-LOOKUP, $x = 8$	0.73	0.03	0.38

to represent the pointers, matches, and literals of an LZ77 compression process [20], and makes use of a set of cascading lookup tables to manage a relatively small alphabet. (The drawback of Gzip in our text database application [4] is, of course, that it is impossible to provide random access into the text using the Gzip sliding window model.)

Table VIII shows the speed obtained by the various methods on Sparc 10 Model 512 hardware, an 80 MIP workstation.[1] The optimal search decoder used to generate the speed figures in the table uses hard-coded constants similar to those shown in Fig. 2. All other methods used loops and explicit references to *base* and *lj_base* when decoding.

As expected, throughput generally improves as the search method becomes more refined, with the fastest times usually being those of the TABLE-LOOKUP approach. Note, however, that the linear search is faster than is a binary search; this is because of smaller loop overheads and, for the nonwords, the low entropy. Finally, note that much of the speed advantage of the optimal search method arises because the numeric values involved are hard coded into the program, and all of the array lookups in *lj_base* are eliminated. Indeed, binary and linear search can be similarly hard coded, and give performance nearly as good as the optimal search; in the case of hard-coded searches, there is, of course, no difference in "loop" overheads among the three methods, and the number of comparisons performed is a good indicator of throughput.

Table IX shows the average number of *code* versus *base* (or *lj_base*) comparisons performed per symbol in each experiment. Note how the TABLE-LOOKUP method takes well under one inspection per symbol on average, as a large fraction of the codewords have their length calculated deterministically from the $x = 8$-bit prefix. Note also that the binary search requires more probes into lj_base than linear search for the low-entropy nonwords distribution. Another interesting observation is that the number of probes for the binary and optimal search techniques for the words distribution are similar, and so hard coding both methods will lead to similar speedups.

Thus far, we have concentrated on the benefits of canonical coding for large alphabets. We also compared the TABLE-LOOKUP decoder with previously described methods using as a test harness a zero-order character-based model. The change of model allows the methods of other authors to be run in reasonable amounts of space, which would not have been possible with a word-based model. For the same test file, we had $n = 96$ distinct symbols, a maximum code length of $L = 22$, and an entropy of 4.91 bits/symbol. Decoding throughput and memory requirements are reported in Fig. 3.

[1]Note that the speeds for method CANONICAL-DECODE are already faster than those previously reported for this hardware [4], [21], [22]; this is because of the use of a 32-bit input token in the compressed bit stream rather than an 8-bit input token. This change was common to all of the methods listed.

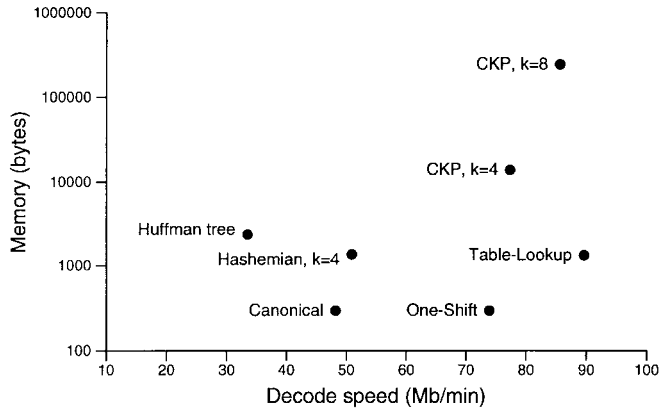

Fig. 3. Resource use during decoding with a character-based zero-order model ($n = 96, L = 22$), assuming a probability-sorted alphabet.

The finite-state machine of Choueka *et al.* [16] is represented in Fig. 3 by the two points marked "CKP." In this method, larger values of k increase throughput, but at the expense of large tracts of memory—over 250 Kbytes, for $k = 8$. More to the point, both the $k = 4$ and $k = 8$ implementations were outperformed in terms of both speed and space by Algorithm TABLE-LOOKUP. The "Huffman tree" data point in Fig. 3 describes the resources required for traditional bit-by-bit traversal of a code tree to decode the source, while the point marked "Hashemian" shows the use of the tree/table method of Hashemian [12] using $k = 4$. Both of these tree-based approaches are outperformed by Algorithm CANONICAL-DECODE, the starting point of our development.

VII. SUMMARY

We have detailed the full sequence of operations needed to undertake minimum redundancy coding. We believe that the key to fast and compact encoders and decoders is to divorce the codewords from the source alphabet by using a mapping that allows the use of ordinal symbol identifiers and a canonical assignment of codewords. In particular, we observe that explicitly tree-based decoding is an anachronism and usually best avoided, despite the attention such methods have received in textbooks, in the research literature, and in postings to the various network news groups.

We have also shown how the speed of canonical decoding can be improved through the use of a left-justified table of *base* values and an auxiliary array *start* to reduce the time spent searching. In combination, these two improvements allow decoding speed to be improved by approximately 50%, and the improved decoder outperforms all of the other proposed methods we are aware of.

One final point that warrants further discussion is the requirement in Algorithm TABLE-LOOKUP that L, the length of a longest codeword, be less than w, some convenient unit of memory. For example, on many computers, $w = 32$ is the largest sensible unit of storage. To guard against pathological codes, length-limited minimum redundancy codes should be used. In collaboration with Katajainen, we have shown elsewhere [23] that length-limited codes can be constructed in only slightly more time and space than conventional minimum redundancy codes. Moreover, once codeword lengths are calculated, encoding and decoding using length-limited codes can be handled in exactly the same manner as are encoding and decoding using unrestricted codes—that is, through the use of the mechanisms described in this paper. We thus argue that *all* minimum redundancy codes can and should be calculated to meet a length limit appropriate for the computer system being used; a length limit is essential if the speed of any implementation in hardware or software is not to be compromised by the need to guarantee the integrity of the decoded data.

References

[1] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, pp. 1098–1101, Sept. 1952.

[2] D. A. Lelewer and D. S. Hirschberg, "Data compression," *Comput. Surveys*, vol. 19, pp. 261–296, Sept. 1987.

[3] J. L. Gailly, "Gzip program and documentation," 1993; available by anonymous ftp from prep.ai.mit.edu:/pub/gnu/gzip-*.tar.

[4] J. Zobel and A. Moffat, "Adding compression to a full-text retrieval system," *Softw.—Pract. Exp.*, vol. 25, pp. 891–903, Aug. 1995.

[5] M. Hankamer, "A modified Huffman procedure with reduced memory requirements," *IEEE Trans. Commun.*, vol. 27, pp. 930–932, June 1979.

[6] J. van Leeuwen, "On the construction of Huffman trees," in *Proc. 3rd Int. Colloquium Automata, Languages, and Programming*, Edinburgh Univ., Scotland, July 1976, pp. 382–410.

[7] J. Katajainen and A. Moffat, "In-place calculation of minimum-redundancy codes," submitted. Preliminary version in *Proc. 1995 Workshop Algorithms and Data Structures*, Kingston, Ont., Canada, Aug. 1995, pp. 393–402. Source code avaliable from http://www.cs.mu.oz.au/~alistair/inplace.c, Feb. 1997.

[8] A. Moffat and A. Turpin, "Efficient construction of minimum-redundancy codes for large alphabets," *IEEE Trans. Inform. Theory*, to be published. Preliminary version in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Mar. 1995, pp. 192–201.

[9] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Commun. ACM*, vol. 7, pp. 166–169, Mar. 1964.

[10] J. B. Connell, "A Huffman-Shannon-Fano code," *Proc. IEEE*, vol. 61, pp. 1046–1047, July 1973.

[11] D. S. Hirschberg and D. A. Lelewer, "Efficient decoding of prefix codes," *Commun. ACM*, vol. 33, pp. 449–459, Apr. 1990.

[12] R. Hashemian, "High speed search and memory efficient Huffman coding," *IEEE Trans. Commun.*, vol. 43, pp. 2576–2581, Oct. 1995.

[13] D. R. McIntyre and F. G. Wolff, "An efficient implementation of Huffman decode tables," *Congressus Numerantium*, vol. 91, pp. 79–92, 1992.

[14] M. A. Bassiouni and A. Mukherjee, "Efficient decoding of compressed data," *J. Amer. Soc. Inform. Sci.*, vol. 46, pp. 1–8, Jan. 1995.

[15] H. Tanaka, "Data structure of the Huffman codes and its application to efficient encoding and decoding," *IEEE Trans. Inform. Theory*, vol. IT-33, pp. 154–156, Jan. 1987.

[16] Y. Choueka, S. T. Klein, and Y. Perl, "Efficient variants of Huffman codes in high level languages," in *Proc. 8th ACM–SIGIR Conf. Inform. Retrieval*, Montreal, Canada, June 1985, pp. 122–130, ACM, NY.

[17] A. Sieminski, "Fast decoding of the Huffman codes," *Inform. Processing Lett.*, vol. 26, pp. 237–241, May 1988.

[18] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Trans. Inform. Syst.*, to be published. Preliminary version in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Mar. 1995, pp. 202–211. Source software available from ftp://munnari.oz.au/pub/arith_coder.

[19] D. Harman, "Overview of the second text retrieval conference (TREC-2)," *Inform. Processing Manage.*, vol. 31, pp. 271–289, May 1995.

[20] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. IT-23, no. 3, pp. 337–343, 1977.

[21] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. New York: Van Nostrand Reinhold, 1994.

[22] A. Moffat, J. Zobel, and N. Sharman, "Text compression for dynamic document databases," *IEEE Trans. Knowledge Data Eng.*, vol. 9, pp. 302–313, Mar. 1997.

[23] J. Katajainen, A. Moffat, and A. Turpin, "A fast and space-economical algorithm for length-limited coding," in *Proc. Int. Symp. Algorithms and Computation*, J. Staples, P. Eades, N. Katoh, and A. Moffat, Eds., Cairns, Australia, Dec. 1995, pp. 12–21, Springer-Verlag, LNCS 1004.

Alistair Moffat received the Ph.D. degree from the University of Canterbury, New Zealand, in 1986.

Since then, he has been a member of the academic staff at the University of Melbourne, Australia, and is currently an Associate Professor. His research interests include text and image compression, techniques for indexing and accessing large text databases, and algorithms for sorting and searching. He is a coauthor of the 1994 book, *Managing Gigabytes: Compressing and Indexing Documents and Images*, and has written more than 70 refereed papers.

Dr. Moffat is a member of the ACM and of the IEEE Computer Society.

Andrew Turpin is a research student at the University of Melbourne, and is currently completing the Ph.D. degree, investigating efficient mechanisms for calculating and employing minimum redundancy codes and length-limited minimum redundancy codes.