Efficient Decoding of Prefix Codes

Daniel S. Hirschberg and Debra A. Lelewer

Abstract

We discuss representations of prefix codes and the corresponding storage space and decoding time requirements. We assume that a dictionary of words to be encoded has been defined and that a prefix code appropriate to the dictionary has been constructed. The encoding operation becomes simple given these assumptions and given an appropriate parsing strategy, therefore we concentrate on decoding. The application which led us to this work constrains the use of internal memory during the decode operation. As a result, we seek a method of decoding which has a small memory requirement.

Introduction

Data compression is an important and much-studied problem. Compressing data to be stored or transmitted can result in significant improvements in the use of computing resources. The degree of improvement that can be achieved depends not only on the selection of a data compression method, but also on the characteristics of the particular application. That is, no single data compression algorithm will be superior in every application. The very meaning of "superior" is application dependent. While the goal of data compression is to represent a message as succinctly as possible, a particular application may modify that goal by placing additional requirements on the performance of the data compression system. In other words, the application may define parameters that guide the selection of the data compression method. These parameters include, for example, knowledge about the type of data to be compressed and constraints on memory usage and execution speed.

The work we describe here is based on a specific data compression application in which: a)

textual data is to be transmitted and received over a communication line, b) decoding must be performed on-line, and c) the amount of memory available during the decode operation is very limited. The encoder in our data compression system is allowed substantial computational resources. It can expend significant time and space to find a compact representation of the source text. Once the representation is constructed, it will be transmitted to the decoder. The decoder may be viewed as a special-purpose translator with very limited space. This space limitation provides an interesting challenge.

The application on which this research is based involves a minimal memory constraint. While technology is providing increasing amounts of memory at low cost, minimizing the use of memory will always be a goal in mass production applications of data compression.

We employ a dictionary compression technique, that is, an algorithm that compresses a source text by replacing strings of characters in the source by pointers to a dictionary. The *dictionary* is a collection of n strings of varying lengths. Long dictionary entries have higher potential for compression than short ones in that we replace a large number of characters with a single codeword. However, we must also take into account the frequency with which a dictionary entry occurs in the source text. We want to assign short codewords to frequently-occurring strings; if a string occurs only rarely its codeword may be too long to provide good compression even though the string being replaced is itself quite long. The degree of compression to be achieved by a dictionary compression system is largely dependent on the choice of the dictionary; however, it is also necessary to represent the pointers efficiently. We choose to represent pointers by prefix codes based on the relative frequencies of the dictionary entries they represent. The Huffman code is the most widely-known prefix code and is minimal in that it provides the best compression of any prefix code applied to a fixed dictionary [Huffman 1952]. Arithmetic codes, which are not prefix codes, can provide better compression than the Huffman code when applied to the same dictionary [Witten, Neal and Cleary 1987]. This improved compression is possible because arithmetic codes are not constrained to map an integer number of bits to each dictionary entry. The additional compression they provide is generally a few percent.

An offsetting advantage of Huffman codes is that they are more robust. While an error in a single bit will prevent the bits that follow from being correctly decoded by an arithmetic decoder, Huffman codes tend to resynchronize quickly, thus localizing damage [Lelewer and Hirschberg 1987]. A more important consideration in terms of the present application is the fact that arithmetic coding uses the frequencies of the dictionary entries during decoding. Our methods do not require the table of frequencies, and as a result we are able to decode

with a much smaller space requirement. For these reasons we elect to use Huffman coding for our application.

The compressed version of the source text consists of a representation of 1) the encoding dictionary, 2) its prefix code and 3) the sequence of codes that can be expanded to recover the original text. Most of the compression is achieved by choosing an appropriate dictionary. The computation of the corresponding prefix code is straightforward. However, the method of representing the dictionary and the prefix code also affects the resulting compression ratio (for moderate-sized files, the representation choice can have a significant impact on the compression ratio). The encoder in our application must construct a representation that is compact and that our space-limited decoder can translate efficiently. The way in which the encoder represents the dictionary and the prefix code is the focus of our work. We partition the encoding dictionary into two parts: 1a) a stream of characters, and 1b) information that permits parsing this stream into individual dictionary entries (e.g., the lengths of the entries or their starting positions). All of our methods prepend the stream of characters to the encoded text and store the characters as part of the decode data structure. It is in the way that 1b) and 2) are represented that the methods differ. We will compare the decode space efficiencies of our methods and the amount of *representation overhead* they incur. We define representation overhead to be the number of bytes in the compressed text used to represent items 1b) and 2). We allow the decoder some limited set-up time to receive the code representation (items 1 and 2) and store the information needed for performing translation. Except for the "lag" due to set-up, the decoder must operate on-line. That is, the time required for decoding must be proportional to the size of the expanded source.

In order that our methods may be presented in the most general form, we define the variables listed in Figure 1. It should be noted that $N \geq \lg n$† bits, that $M \leq V$, that $B \leq V$, and that $A \geq C$ since we must be able to access any dictionary entry with an address. Figure 2 presents a small example dictionary, which we use to illustrate our methods.

Previous Methods

A number of papers have appeared on the subject of implementations of Huffman encoding and decoding. These implementations apply to any prefix code. The more recent of these papers [Sieminski 1988; Choueka et al. 1986] concentrate on fast implementations and reduce processing time by avoiding manipulation of individual bits. However, a price is paid for the reduced time requirements in the form of increased memory requirements.

† lg denotes the base 2 logarithm

symbol	storage requirement for	typical value
A	an address	2 bytes
C	number of characters in a dictionary entry	1 byte
N	an integer between 1 and $n+1$	2 bytes
M	number of codewords of a given length	1 byte
B	length of a codeword (in bits)	1 byte
V	value of a codeword	2 bytes
	meaning	
L	$max - min + 1$	13
max	length of longest codeword (in bits)	12–16
min	length of shortest codeword (in bits)	1–3

Figure 1 Variables used to define storage requirements.

string	frequency
$abcd$	10
rst	9
$wxyz$	15
qu	7
lm	2
ps	2
the	22

Figure 2 An example dictionary.

Sieminski's method requires 64 K bytes to store the decode tables for a simple situation in which the dictionary contains only 127 individual characters. The size of the decode tables grows exponentially if dictionary entries longer than one character are used [Sieminski 1988]. The method of [Choueka et al. 1986] requires $O(n^2)$ extra space where n is the number of dictionary entries. While processing time is of concern, our primary criterion is the efficient use of internal memory during decoding. Thus these methods are inappropriate for our

purposes.

Hankamer [1979] describes a modified Huffman procedure with reduced memory requirements. The reduced memory requirements are attained by reducing the size of the dictionary and computing a suboptimal Huffman code. Hankamer's method assumes fixed-length dictionary entries and there is no obvious extension to variable-length entries. This fact coupled with the loss of optimality renders the method inappropriate for our needs. Tanaka [1987] gives a finite automaton-based Huffman decoding algorithm. His method assumes single character dictionary entries. A straightforward modification to allow for variable-length entries is similar to our Method A1 (which follows) in terms of execution speed, but requires approximately 67% more memory.

Method A

Our first solution to the problem of decoding in restricted memory uses the Huffman code tree to represent the dictionary. However, we do not use the obvious linked implementation in which each internal node contains pointers to its left and right subtrees; the space requirements of this implementation are prohibitive. Instead, Method A1 employs an implicit representation of the tree structure. Method A2 is a variation of Method A1 that provides improved storage utilization.

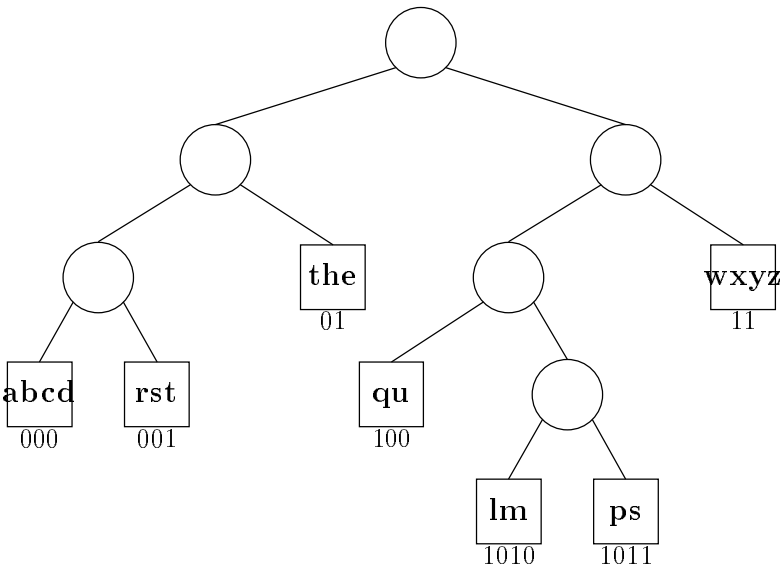

Figure 3 A Huffman tree for the example dictionary.

Method A1

Method A1 uses a total of $nC + (n-1)A$ space in addition to the space required for the n dictionary entries (the space for a dictionary entry is the space required to store the characters that make up the entry). The code representation and the dictionary are stored as a single structure. The prefix code is represented by the corresponding binary tree stored in preorder form. Preorder storage is defined recursively: the root node is stored first, followed by its left subtree stored in preorder form, and then its right subtree in preorder form. In our storage scheme, a leaf node contains a flag bit (set to one, distinguishing between internal nodes and leaves), the length of the corresponding dictionary entry, and the entry itself. For each internal node we store two items, a flag bit (set to zero) and an address. The address component of an internal node is the address of its right subtree. The left subtree for an internal node is stored immediately following the node itself. A tree with n leaves contains $n-1$ internal nodes. Thus, the total storage in addition to the dictionary entries is nC for the leaf nodes and $(n-1)A$ for the internal nodes, assuming that there is a spare bit in the address and length fields. In our application, for which the typical values given in Figure 1 apply, the storage requirement is $3n-2$ bytes. Figure 3 shows a Huffman tree for the example dictionary. The codeword for each dictionary entry appears under the entry. We use the convention that left branches are labeled '0' and right branches '1'. Figure 4 gives the corresponding decode data structure. We represent tree nodes as tuples of the form (0,address) or (1,length,entry). The address values are based on allowing 2 bytes for an address ($A = 2$) and 1 byte for each character and each string length ($C = 1$). We assume that the first bit of an address or length field stores the flag bit.

The storage scheme described above allows for simple decoding. For each codeword we begin at the first position of the decode table and we decode one bit at a time. On a 0 bit we move from an internal node to its left child by advancing over the address field. On a 1 bit we use the address field to move to the right subtree of the current internal node. We continue to decode bits until a flag value of 1 is encountered, indicating a leaf node. At this point we have detected the end of a codeword and located the corresponding dictionary entry. The dictionary entry is appended to the decoded output, and we return to the first position of the decode table ready to decode the next codeword. The following operations are performed for each codeword in the encoded source. We use $address(k)$ to represent the address component of an internal node k, $flag(k)$ to represent the flag component of any node k, and $length(k)$ to represent the length component of a leaf node k.

$k \leftarrow 1$

repeat

receive bit
if $bit = 0$
then $k \leftarrow k + A$
else $k \leftarrow address(k)$
$flag_value \leftarrow flag(k)$
until $flag_value = 1$
append contents of memory locations $k \ldots k + length(k) - 1$ to the decoded output

The encoder transmits the tree to the decoder in the form we have described. Thus, the representation overhead associated with Method A1 is $nC + (n-1)A$, and the "lag" time consists of the time necessary to receive and store the tree.

$address$	1	3	5	7	12	16	20
$contents$	$(0, 20)$	$(0, 16)$	$(0, 12)$	$(1, 4, abcd)$	$(1, 3, rst)$	$(1, 3, the)$	$(0, 35)$
$address$	22	24	27	29	32	35	
$contents$	$(0, 27)$	$(1, 2, qu)$	$(0, 32)$	$(1, 2, lm)$	$(1, 2, ps)$	$(1, 4, wxyz)$	

Figure 4 Method A1 storage of example dictionary.

Method A2

The storage requirement of Method A1 can be improved in some cases by exploiting the fact that the length values need not be stored in the decode data structure. The key observation that allows us to eliminate the string lengths is that we can find the length of an entry by subtracting its starting address from the starting address of its preorder successor. The starting address of any leaf node's preorder successor can be found easily, trivially, in fact, if the leaf, x, is a left child of its parent. In this case, the preorder successor of x is its sibling, and the address of the sibling is stored in x's parent node. In the other case, when x is a right child, we can walk from x to its preorder successor as follows: we walk up '1' branches until we reach a node that is not a right child; at this point, we walk up a single '0' branch and then down a '1' (right) branch. In other words, the preorder successor of x is the right child of the lowest internal node from which we follow a '0' (left) branch to x. This characterization is also valid when x is a left child, since x's parent is the lowest internal node from which we follow a left branch to x; and x's preorder successor is the right child of this (parent) node. The only node for which the above characterization is not valid

is the final node in the preorder listing. This node lies on a path from the root consisting of only right branches, and it has no preorder successor. So that we can decode this final node, we store the address of its (nonexistent) preorder successor in address 0 of the decode data structure, ahead of the preorder representation of the decode tree. Thus we store n addresses in Method A2 instead of the $n-1$ addresses used in Method A1.

In the Method A1 data structure, address values are coupled with flag bits to represent internal nodes, and length values are coupled with flag bits and combined with character strings to represent leaf nodes. The coupling is accomplished by using the leading bit of the address or length value for storing the flag. In eliminating the length value from a leaf node, we are presented with the problem of how to store the flag bit. The best solution to this problem is to couple the flag bit with the leading character of the dictionary entry. In order for this to be possible, we must be able to store characters in $b-1$ bits (where b is the number of bits per byte). This assumption may be reasonable on machines with 8-bit bytes where the application involves storing or transmitting text. The printable characters typical of many text files can be represented in seven bits. Under this assumption, the storage requirement of Method A2 becomes nA, as compared with $(n-1)A+nC$ for Method A1. Using the typical values given in Figure 1, we have $2n$ bytes for Method A2, as compared with $3n-2$ bytes for Method A1.

If the assumption of a spare bit in character storage is not valid, eliminating the lengths may not provide an improvement in storage utilization. Since high-level languages have the byte as the atomic unit of addressable memory, we are forced to store the flag in a byte when neither the length field nor the character field can accommodate it. If string lengths can be stored in a single byte ($C=1$) with a spare bit, we gain nothing by storing a one-byte flag instead of a one-byte (flag,length) pair. In fact, the storage requirement for Method A2 would be $nA+n$ bytes as compared with $(n-1)A+n$ bytes for Method A1. However, in a case where lengths require more than one byte of storage ($C \geq 2$), the one-byte flag would be an improvement over the C-byte (flag,length) pair. In this case, Method A1 requires $(n-1)A+nC$ bytes of storage and Method A2 requires only $nA+n$. In addition, the use of the (flag,length) coupling depends on the assumption that lengths can be stored in such a way as to provide a spare bit for the flag. If this assumption is not valid, storing the flag alone will provide a space improvement over storing the (flag,length) pair in $C+1$ bytes. In summary, the elimination of the length values from the Method A1 data structure is not guaranteed to provide improved storage utilization, but does so under fairly general conditions. In fact, Method A2 will be superior to Method A1 unless characters require all b bits in a byte and string lengths require at most $b-1$ bits. And in this case Method A2

can lose by most $A + 1$ bytes!

In Figure 5 we give the Method A2 data structure for the example dictionary of Figure 2 under the assumption that each character contains a spare bit that can be used for the flag value. We assume that address fields also contain the spare bit and that $A = 2$. We represent internal nodes as (flag,address) pairs and leaf nodes as (flag,entry) pairs.

$address$	0	2	4	6	8	12	15	18
$contents$	34	$(0, 18)$	$(0, 15)$	$(0, 12)$	$(1, abcd)$	$(1, rst)$	$(1, the)$	$(0, 30)$
$address$	20	22	24	26	28	30		
$contents$	$(0, 24)$	$(1, qu)$	$(0, 28)$	$(1, lm)$	$(1, ps)$	$(1, wxyz)$		

Figure 5 Method A2 storage of example dictionary of Figure 2.

Using the Method A2 data structure to decode is very similar to using the Method A1 structure. The only difference is that, in addition to the address of the dictionary entry being decoded, we are also looking for the address of its preorder successor. The following instructions are performed for each codeword. We use $address(k)$ and $flag(k)$ as in Method A1; p represents the current candidate for the address of the preorder successor. We use the notation $contents(0)$ to retrieve the successor of the last node in the preorder listing from memory location 0. Decode speed is very similar to that of Method A1; the only extra time is due to storing an address in p for each 0 bit.

$p \leftarrow contents(0)$
$k \leftarrow 2$
repeat
    receive bit
    if $bit = 0$
    then $p \leftarrow address(k)$
        $k \leftarrow k + A$
    else $k \leftarrow address(k)$
    $flag_value \leftarrow flag(k)$
until $flag_value = 1$
append contents of memory locations $k \ldots p - 1$ to the decoded output

The encoder transmits the tree to the decoder in the form we have described. Thus, as-

suming a spare bit in character bytes, the representation overhead for Method A2 is nA and the set-up time consists of the time necessary to receive and store the tree. Both representation overhead and set-up time are smaller for Method A2 than for Method A1. Figures 10 and 11 present space and time comparisons of our methods. The data for Method A1 presumes the spare bit in the address and length bytes, and for Method A2 the spare bit in character bytes is assumed.

Method B

The second method we discuss is based on the concept of a canonical Huffman code defined by Schwartz and Kallick [1964] and by Connell [1973]. We describe this concept first and then our implementation of it. The essence of the canonical code concept is that Huffman's algorithm is needed only to compute the lengths of the codewords to be mapped to the dictionary entries. Once lengths are determined, actual codewords may be specified in many ways; the only necessary condition is that they satisfy the prefix property. This is true for prefix codes in general. Intuitively, the canonical code may be viewed as one that builds the prefix code tree from left to right in increasing order of depth (i.e., codeword length) with the convention that each leaf is placed at the "first" position (from left to right) available to it. The example dictionary has codeword length sequence [2,2,3,3,3,4,4]. In constructing the canonical code, the first codeword of length two is placed at the left edge of level two of the tree. Using the convention that left branches are labeled with 0 and right branches with 1, the first codeword is 00. The second codeword of length two is the sibling of the first, 01. The first codeword of length three is placed at the first available position on level three of the tree. Level three is filled from left to right by placing codewords 100, 101, and 110. The length-four codewords, 1110 and 1111, complete the tree. The canonical code tree for the example dictionary is given in Figure 6. The codeword for each dictionary entry appears under the entry.

The canonical code possesses some nice mathematical properties. The codewords of a given length are consecutive binary numbers. The first codeword of length l, c_l, is related to the last codeword of length $l-1, d_{l-1}$, by the equation $c_l = 2(d_{l-1}+1)$. In other words, the first codeword of length l is obtained from the last codeword of length $l-1$ by adding 1 to the binary number represented by d_{l-1} and shifting that binary number left once. In the case where some lengths are unused, as in [1,3,3,3,4,4], the codewords of length 3 are consecutive binary numbers as are the codewords of length 4. The function that computes the first length-3 codeword from the length-1 codeword is $2(2(d_1+1))$; that is, to move down two levels in the tree from level 1 to level 3, two shifts are required. For the length sequence

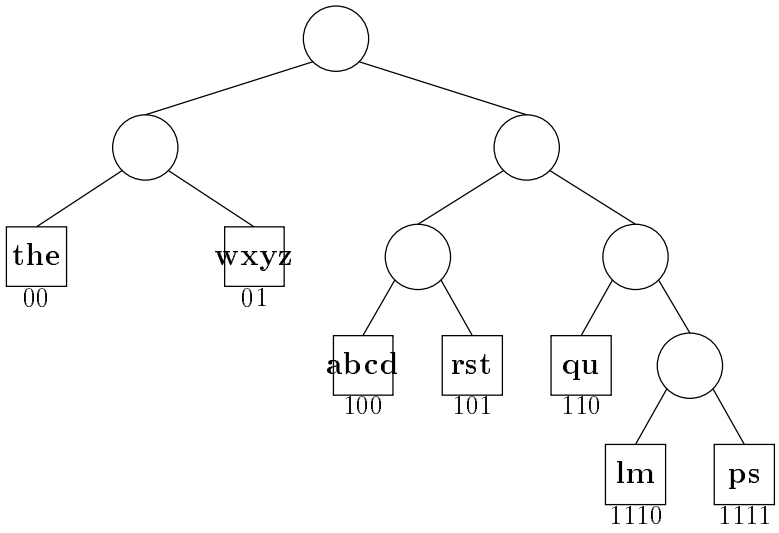

Figure 6 The canonical Huffman code tree for the example dictionary of Figure 2.

[1,3,3,3,4,4], the canonical code is $\{0, 100, 101, 110, 1110, 1111\}$. Every canonical code has a string of zeros as its first (shortest) codeword and a string of ones as its last (longest) codeword. We say that a canonical code has the *numerical sequence property.*

We now discuss the way in which the numerical sequence property contributes to reducing memory requirements. First, the canonical code eliminates the need for the encoder to transmit to the decoder an explicit representation of the tree; the length sequence is sufficient to define the tree. We represent the length sequence as a list consisting of 1) *min*, the length of the shortest codeword, 2) *max*, the length of the longest codeword, and 3) the number of codewords of each length. The first example above is, thus, represented by 2,4,2,3,2 and the second by 1,4,1,0,3,2. In most cases this representation is more compact than a list of the lengths of all of the codewords. If the encoder uses the length list to define the code, the size of the representation is $2B + LM$ where $L = max - min + 1$; M represents the number of bytes required to store the maximum number of codewords of any given length and B the number of bytes required to store the length of a codeword. We will show that the data structure needed by the decoder can be constructed efficiently given the length list.

In addition to providing a compact representation of the code, the numerical sequence property may be used to index into the data dictionary. This is done through the use of two small tables, *limit* and *base*. Each of these tables is indexed from *min* to *max*. The *limit* table is used in decoding to detect the end of a codeword. The entry $limit[i]$ contains

the value of the largest codeword of length at most i. The numerical sequence property guarantees that the numerical value of a codeword of length i is greater than the value of any shorter codeword. Thus if the binary value of a string of i bits is greater than $limit[i]$ the string is not a codeword but a prefix of a codeword. The decoder reads min bits from the coded text. If the binary value of this bit string is less than or equal to $limit[min]$ the bit string represents a codeword. If the value of the first min bits is greater than $limit[min]$ the decoder reads another bit, updates the value of the bit string, and compares that value to $limit[min+1]$. This process continues until the value of the bit string of length i is less than or equal to $limit[i]$ for some i. At this point we have recognized a codeword. Once the end of a codeword is detected the *base* table may be used to locate the corresponding dictionary entry. The *base* table as defined in [Connell 1973] maps a codeword value onto the relative position of the corresponding dictionary entry in a list of dictionary entries.

The information provided by the *limit* and *base* tables is sufficient to allow decoding if the entries of the data dictionary are all of the same length; however for variable-length entries we need the address of the appropriate entry, not an index. We present two solutions to this problem. We comment that tables *limit* and *base* as defined by Connell [1973] are redundant with respect to one another. That is, the information contained in the *base* table can be extracted from the *limit* table entries. However, the *base* table can be represented in very little space, and contributes substantially to the clarity of exposition of our methods. Eliminating the *base* table also results in slower decoding; therefore we maintain the *base* table.

Method B1

Method B1 adapts Connell's *base* table method to allow for variable-length dictionary entries by introducing an *address* table indexed from 1 to $n+1$. The value of $address[k]$ is the address of the first character of the k^{th} dictionary entry. The entries are stored in a *string* table that is organized in the following way: entries are stored in nondecreasing order by codeword length and the block of entries with codeword length i is stored in order of decreasing codeword value. In terms of the prefix tree we store the dictionary in *modified level order*; that is, in increasing order by level and in order from right to left on each level (of course we are storing only the leaves of the prefix tree). The *base* table provides pointers into the *address* table; that is, $base[i]$ contains x such that $address[x]$ is the starting address of the block of dictionary entries with codeword length i. When a codeword c of length i is recognized, $limit[i] - value(c)$‡ provides an offset in the list of codewords of length i.

‡ $value(c)$ is the binary value of codeword c

Thus $p = base[i] + limit[i] - value(c)$ is the subscript in the *address* table at which the beginning of the corresponding dictionary entry is stored. The length of the entry is given by $address[p+1] - address[p]$. The address and length of the entry are all we need to append the entry to the output of the decoder. The storage requirement at decode time consists of LV for the *limit* table (*limit* contains codeword values), LN for the *base* table (*base* contains subscripts from 1 to $n+1$), and $(n+1)A$ for the *address* table. In most cases we expect $LV + LN + (n+1)A$ to be an improvement over the $nC + (n-1)A$ requirement of Method A1. In practice L is generally $O(\lg n)$ while a "typical" value of L is 13. Therefore Method A1 requires $3n-2$ bytes and Method B1 $2n+54$ in a typical application. The storage requirement of Method B1 will always be greater than the $2n$ requirement of Method A2; thus, Method B1 provides no improvement in an application in which character bytes contain an unused bit. In terms of translation time, Method B1 is expected to be a little bit slower than the A Methods, but not significantly slower.

The encoder transmits the length list, the strings, and their lengths as a preface to the encoded text. Thus, the representation overhead is $2B + LM + nC$. The representation is transmitted in the following form: first, *min* and *max*; then for each codeword length i (from *min* to *max*), n_i followed by n_i $(length, str)$ pairs. Each n_i represents the number of dictionary entries with codeword length i and each $(length, str)$ pair gives the number of characters in a dictionary entry followed by the character string itself. The entries with codeword length i are listed in order of decreasing codeword value. The decoder performs the following calculations to set up the decode data structure. In addition to the time required to receive the data, the decoder performs $\theta(n)$ operations in setting up the *address* table and $\theta(L)$ operations in constructing tables *limit* and *base*.

```
s ← 1
a ← 1
receive min, max
for i ← min to max do
        receive n_i
        if i = min
        then base[min] ← 1
             limit[min] ← n_min − 1
        else base[i] ← base[i − 1] + n_{i−1}
             limit[i] ← 2(limit[i − 1] + 1) + n_i − 1
        for j ← 1 to n_i do
                receive length, str
                store str in string[s ··· s + length − 1]
                address[a] ← s
```

$a \leftarrow a + 1$
$s \leftarrow s + length$
endfor
endfor
$address[a] \leftarrow s$

Figure 7 gives the Method B1 data structure for the example dictionary. The addresses represent byte addresses of dictionary entries; we assume that the starting address is 1, and that each character of an entry occupies 1 byte. Figures 10 and 11 provide space and time comparisons of Method A1, Method A2, and Method B1.

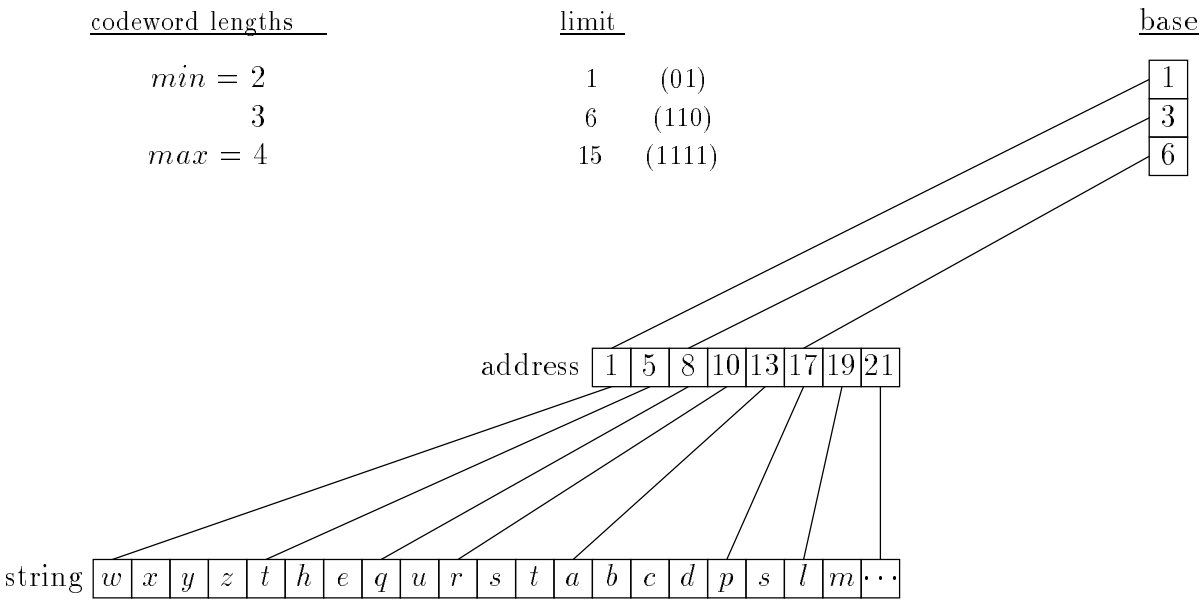

Figure 7 Method B1 data structure for the example of Figure 2.

Method B2

We now present a modification of Method B1 that can provide space utilization superior to that of Method A2. Method B2 is actually a collection of methods, parameterized by a variable k. The time-space compromise that best fits the requirements of a particular application can be selected by fixing an appropriate value of k. The improvement in Method B2 over Method B1 is achieved by storing fewer than n address values; the value

of the parameter k determines what fraction of the n address values are stored. Method B2 uses the *limit* and *base* tables exactly as in Method B1. The dictionary is represented by three tables. The first table, *string*, contains the dictionary entries stored as in Method B1 (i.e., in modified level order). The second table, *address*, is indexed from 1 to $\lfloor \frac{n}{k} \rfloor$ and stores the address of every k^{th} dictionary entry, with $address[j]$ containing the address of entry jk. The third table, *len*, is indexed from 1 to $n - \lfloor \frac{n}{k} \rfloor$ and contains string lengths. Thus the space requirements of Method B2 are: $LV + LN$ for the *limit* and *base* tables, $\lfloor \frac{n}{k} \rfloor A$ for the *address* table, and $(n - \lfloor \frac{n}{k} \rfloor)C$ for the *len* table.

The *limit* table is used to recognize codewords as in Method B1. The *base* table again yields an index into the list of dictionary entries; if $base[i] = x$ then the x^{th} dictionary entry is the first entry (in modified level order) with codeword of length i. When a codeword c of length i has been decoded, we use $p = base[i] + limit[i] - value(c) - 1$ to find the corresponding dictionary entry. If $p \bmod k = 0$, the address of the first character of the entry is stored in $address[\frac{p}{k}]$. If $p \bmod k \neq (k-1)$, the length of entry p is stored in $len[p - \lfloor \frac{p}{k} \rfloor + 1]$. Thus, when $p \bmod k = 0$, both the address and the length of the corresponding dictionary entry are stored in the decode data structure. When $p \bmod k \neq 0$, $address[\lfloor \frac{p}{k} \rfloor]$ is a pointer to the block of k entries that includes the one we seek. We "walk" along this block until we find the entry corresponding to c. This walk can be viewed as a sequence of "jumps" that use the *len* values to jump over entries. The number of jumps is given by $p \bmod k$; the maximum number of jumps is $k - 1$. If $p \bmod k \neq (k-1)$, the length of the entry is stored in the *len* table; otherwise, the length of the entry is computed from the starting address of its successor in the modified level order listing (i.e., $address[\lfloor \frac{p}{k} \rfloor + 1]$). The following calculations provide the starting address *start* and the *length* corresponding to any index p.

$p \leftarrow base[i] + limit[i] - value(c) - 1$
$q \leftarrow \lfloor \frac{p}{k} \rfloor$
if $q = 0$
then $start \leftarrow 1$
else $start \leftarrow address[q]$
$r \leftarrow p \bmod k$
$t \leftarrow p - q$
for $i \leftarrow 1$ to r do
    $start \leftarrow start + len[t - i + 1]$
endfor
if $r \neq k - 1$
then $length \leftarrow len[t + 1]$
else $length \leftarrow address[q + 1] - start$

As in Method B1, the encoder transmits the length list, the strings, and their lengths. Thus the representation overhead is $2B + LM + nC$. Tables *limit* and *base* are built exactly as in Method B1. The following code includes the computations for tables *len* and *address*. The set-up time is again $\theta(n) + \theta(L)$.

```
s ← 1
a ← 1
l ← 1
count ← 0
receive min, max
for i ← min to max do
        receive n_i
        if i = min
        then base[min] ← 1
             limit[min] ← n_min − 1
        else base[i] ← base[i − 1] + n_{i−1}
             limit[i] ← 2(limit[i − 1] + 1) + n_i − 1
        for j ← 1 to n_i do
             receive length, str
             store str in string[s ··· s + length − 1]
             if count mod k ≠ k − 1
             then len[l] ← length
                  l ← l + 1
                  if count ≠ 0 and count mod k = 0
                  then address[a] ← s
                       a ← a + 1
             count ← count + 1
             s ← s + length
        endfor
endfor
if count mod k = 0
then address[a] ← s
```

Figure 8 gives the Method B2 data structure for the example of Figure 2 with $k = 2$. A comparison with the other methods is provided in Figures 10 and 11. We note that if $k = 1$ the storage requirement for Method B2 reduces to the requirement for Method B1.

We provide a second example for Method B2 in Figure 9. The data structure for an example with a larger dictionary and $k = 3$ is given. The reader can use the *limit* table values to verify that the codewords for {*wxyz*, *the*, *qu*, *rst*, *abcd*, *ps*, *lm*, *out*, *rt*} are {0, 110, 101, 100, 11110, 11101, 11100, 111111, 111110}.

codeword lengths	limit		base	len
$min = 2$	1	(01)	1	4 $(wxyz)$
3	6	(110)	3	2 (qu)
$max = 4$	15	(1111)	6	4 $(abcd)$
				2 (lm)

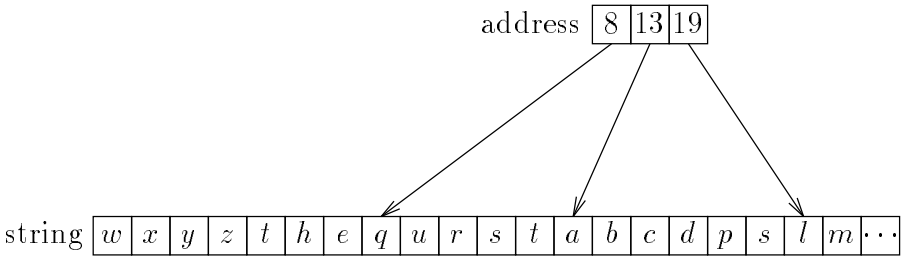

Figure 8 Method B2 data structure for the example dictionary of Figure 2 ($k = 2$).

The parameter k determines the decode speed of Method B2 as well as its storage requirement. The maximum number of jumps determines the worst case time for appending one dictionary entry to the output. The maximum number of jumps is $k-1$. It is important to recognize that the time-space tradeoff provided by Method B2 is nonlinear. When $k = 1$, Method B2 stores n addresses; when $k = n$, Method B2 stores 1 address and $n - 1$ lengths. Assuming $A = 2$ and $C = 1$, the choice $k = 1$ requires $2n$ bytes of storage, and the choice $k = n$ requires $n + 1$ bytes. When $k = 2$, the storage requirement is $1.5n$ bytes, essentially midway between the requirement for $k = 1$ and that for $k = n$. However, the choice of $k = 2$ may result in decode speed much closer to that provided by $k = 1$ than that provided by $k = n$. The extra decode time required by Method B2 (as compared to Method B1) is proportional to the number of jumps. When $k = n$, only one address is stored. Thus, the first codeword (in modified level order) can be decoded with no jumps, the second requires 1 jump, and in general the j^{th} requires $j-1$ jumps; the maximum number of jumps required to decode a single codeword is $n - 1$. Employing Method B2 with $k = 2$ reduces the maximum number of jumps to just one. If we compare the use of $k = n$ with the use of $k = 1$, we see that by doubling the space requirement we eliminate the need to jump since every address is stored; however, we can reduce the maximum number of jumps to one at a cost of only

codeword lengths	limit		base	len
$min = 1$	0	(0)	1	4 ($wxyz$)
2	1			3 (the)
3	6	(110)	2	3 (rst)
4	13			4 ($abcd$)
5	30	(11110)	5	2 (lm)
$max = 6$	63	(111111)	8	3 (out)

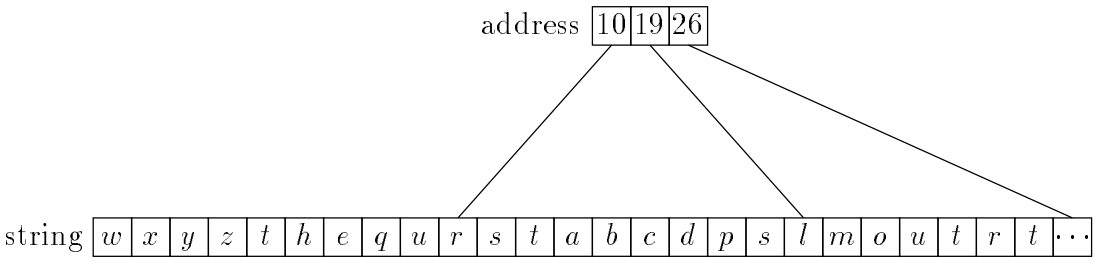

Figure 9 Method B2 data structure for an example with $k = 3$.

50% extra space. In general, a space increase of $\frac{1}{k}$ of the $k = n$ requirement (which stores only a single address and all n string lengths) imposes a ceiling of $k - 1$ on the number of jumps. In practice a k value of about 4 or 5 is reasonable.

Method	Representation Overhead	Decode Space Requirements	Decode Space in "Typical" Application
A1	$(n-1)A + nC$	$(n-1)A + nC$	$3n - 2$
A2	nA	nA	$2n$
B1	$2B + LM + nC$	$(n+1)A + LV + LN$	$2n + 54$
B2	$2B + LM + nC$	$\lfloor \frac{n}{k} \rfloor A + (n - \lfloor \frac{n}{k} \rfloor)C + LV + LN$	$1.2n + 52$

Figure 10 Space comparison of methods.

Method	Receiving Time for Code Description	Additional Set-up Time	Relative Decode Time
A1	$(n-1)A + nC$	*none*	*very fast*
A2	nA	*none*	*very fast*
B1	$2B + LM + nC$	$c_1 L + c_2 n$	*very fast*
B2	$2B + LM + nC$	$c_1 L + c_2 n$	*fast*

Figure 11 Time comparison of methods.

We present a summary of the performance of our methods in Figures 10 and 11. The "typical" values are those given in Figure 1 with the addition of $k = 5$. In the second column of Figure 11, labeled "Receiving Time for Code Description", we give the number of bytes transmitted for the code description; clearly the time required to receive the data is proportional to its size. In column three of Figure 11, c_1 and c_2 represent small constants. We note that, while the A Methods require no additional set-up time, their code descriptions are almost guaranteed to be longer than those of the B Methods, so that the larger receiving time requirement offsets the savings in set-up time.

Additional Implementation Considerations

Reducing transmission time

We consider several issues associated with the representation of: 1a) the stream of characters; 1b) information needed to reconstruct the dictionary from the character stream; and 2) the prefix code. Our discussions have focused on the way in which the representation is stored in the decoder and the way in which it is used to decode the message. We now make some observations on the way in which it is transmitted.

We have assumed that the characters of the dictionary are stored one character per byte in our decode data structures. It is not necessary to respect byte boundaries in transmitting the stream of characters. The stream of characters may be represented in 7- or 8-bit ASCII; however, if the dictionary is very large, it may be significantly more efficient to employ a variable-length coding technique. The canonical Huffman code can be used at very low cost for encoding single characters; only tables *limit* and *base* and an array of characters in modified level order are required for decoding.

In Figures 10 and 11 we include nC bytes in the representation overhead for the lengths of the dictionary entries. We observe, first, that it is not necessary that an integer number of

bytes be used to transmit a string length. In addition, if the lengths of the entries vary across a wide range, we can do much better than nC bytes by using a variable-length representation of the integers such as the Fibonacci codes described by Apostolico and Fraenkel [1987]. If dictionary-entry lengths vary from l_1 to l_2, a fixed-length representation requires $\lg l_2$ bits for each length. The variable-length codes represent small lengths in fewer than $\lg l_2$ bits, but large length values require more bits. The variable-length code is justified, then, if dictionary entries are short on average. For Methods B1 and B2, in which the prefix code is represented by a length list, the same variable-length coding can be applied to codeword lengths. Codeword lengths are expected to be short; it is likely that most of them can be represented in less than one byte. The Fibonacci codes are simple to encode and decode in-place, and are well-suited for representing integers.

Reducing decode time

Another implementation detail worthy of mention is one that can reduce decode time for Method B2. Just as the canonical Huffman code can be viewed as a refinement of standard Huffman coding (in that it selects a particular code tree among multiple optimal trees), we present a further refinement of the canonical code, which we call the *B2-optimal canonical code*. We note, first, that while the canonical code specifies a code tree, it leaves open the question of how to assign the n_i codewords of length i to the n_i dictionary entries. We specify this assignment so as to minimize the average number of jumps (thus a B2-optimal canonical code is one that minimizes decode time).

The B2-optimal code depends on the parameter k and on the interplay between k and the number of codewords of each length. Figure 8 shows that decoding any of **wxyz**, **qu**, **abcd**, or **lm** requires no jumps and that decoding either **the**, **rst**, or **ps** requires one jump. The B2-optimal code reverses the positions of **wxyz** and **the** so that the entry with higher frequency can be decoded without jumps. Two of the level-three entries can be decoded without jumps; these should be the two with highest frequencies. Therefore, **qu** is placed at the middle position of level three and the positions of **abcd** and **rst** are arbitrary. Since **ps** and **lm** have equal frequency their relative positions on level four are arbitrary. The B2-optimal tree typically reduces the average number of jumps in decoding a source text by 25–30%. There are no disadvantages to the use of the B2-optimal tree for decoding; the only cost is the time it takes the encoder to construct the optimal tree rather than an arbitrary canonical tree, and this cost is small.

Summary

Four methods of decoding prefix codes in limited space have been presented. The methods are partitioned into two categories based on the data structuring strategy employed. Method A2 is almost always superior to Method A1; however, the choice among Methods A2, B1, and B2 is less obvious. Parameters of a particular application will influence this decision. Tables comparing time and space requirements of the four methods expose the relevant parameters. The methods we describe define only the decoding phase of a data compression system. The choice of a fixed encoding dictionary is the most critical factor in determining the performance of a system based on our methods. While the representation of the code contributes to the compression ratio attained, most of the compression is achieved by selecting a dictionary that is well-suited to the data being compressed. The size of the dictionary (i.e., number of strings, n) determines the exact time and space requirements of each method. With an advantageous choice of dictionary, our methods can attain compression performance comparable to state-of-the-art techniques such as the Unix utility *compress*. Defining a dictionary that guarantees good compression is, however, a difficult task. When the use of main memory is a concern and sufficient preprocessing time to construct the dictionary is available, our methods provide a solution to the problem of decoding in limited space without sacrificing compression performance. The methods are described in sufficient detail to allow practitioners to implement them easily.

REFERENCES

Apostolico, A. and Fraenkel, A. S. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Trans. Inf. Theory 33*, 2 (Mar. 1987), 238–245.

Choueka, Y., Fraenkel, A. S., Klein, S. T., and Perl, Y. Huffman coding without bit-manipulation. Tech. Rep. CS86-05, Weizmann Institute of Science, Dept. of Applied Mathematics, Rehovot, Israel, 1986.

Connell, J. B. A Huffman-Shannon-Fano code. *Proc. IEEE 61*, 7 (Jul. 1973), 1046–1047.

Hankamer, M. A modified Huffman procedure with reduced memory requirements. *IEEE Trans. Comm. 27*, 6 (Jun. 1979), 930–932.

Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proc. IRE 40*,9 (Sept. 1952), 1098–1101.

Lelewer, D. A. and Hirschberg, D. S. Data compression. *ACM Comput. Surv., 19*, 3 (Sept. 1987), 261–296.

Schwartz, E. S., and Kallick, B. Generating a canonical prefix encoding. *Commun. ACM 7*, 3 (Mar. 1964), 166–169.

Sieminski, A. Fast decoding of the Huffman codes. *Inf. Process. Lett. 26*, 5 (May 1988), 237–241.

Storer, J. A. *Data Compression Methods and Theory*, Computer Science Press, Rockville, Md., 1988.

Tanaka, H. Data structure of Huffman codes and its application to efficient encoding and decoding. *IEEE Trans. Inf. Theory 33*, 1 (Jan. 1987), 154–156.

Witten, I. H., Neal, R. M., and Cleary, J. G. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June 1987), 520–540.