Efficient Variants of Huffman Codes in High Level Languages

Y. Choueka[1], S.T. Klein[1,2], Y. Perl[1,3]

[1]Institute for Information Retrieval and Computational Linguistics (The Responsa Project) and Department of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan, Israel.

[2]Current address:
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot 76100, Israel.

[3]Current address:
Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, USA.

ABSTRACT

Although it is well-known that Huffman Codes are optimal for text compression in a character-per-character encoding scheme, they are seldom used in practical situations since they require a bit-per-bit decoding algorithm, which has to be written is some assembly language, and will perform rather slowly. A number of methods are presented that avoid these difficulties. The decoding algorithms efficiently process the encoded string on a byte-per-byte basis, are faster than the original algorithm, and can be programmed in any high level language. This is achieved at the cost of storing some tables in the internal memory, but with no loss in the compression savings of the optimal Huffman codes. The internal memory space needed can be reduced either at the cost of increased processing time, or by using non-binary Huffman codes, which give sub-optimal compression. Experimental results for English and Hebrew text are also presented.

1. Introduction and Motivation

In most on-line retrieval systems, large files which are stored in secondary memory are frequently accessed. These files should be stored in compacted form, not only to save space, but also to reduce the response-time: the time spent on decompressing is generally largely compensated for by the savings in the number of I/O-accesses, since more information can be read in a single input operation. A given file can be compressed by exploiting its redundancies. For example, if the file is a text written in any natural language, one can devise a special code for its characters, taking into account their distribution and assigning shorter codewords to frequently used letters than to rare ones.

An algorithm for the construction of an optimal code for a given distribution was devised by Huffman [1]. An efficient implementation of the encoding algorithm in time $O(N \log N)$, where N is the size of the encoded alphabet, is given in Even [2]. However, in the context of information retrieval systems, compression is done only once (when building the database), whereas decompression directly affects the response time for on-line queries. We are thus more concerned with a good *decoding* procedure. In spite of their optimality, Huffman codes are not very popular with programmers and are seldom used in practice as they require bit-manipulations and are thus not suitable for smooth programming and efficient implementation in most high-level languages.

The main goal of this paper is to increase the attractivity of Huffman codes, by designing a decoding routine that directly processes only bit-blocks of fixed and convenient size (typically, but not necessarily, integral bytes), making it therefore faster and suitable to high-level languages programming, while still being efficient in terms of space requirements. In principle, byte-decoding can be achieved by using special-built

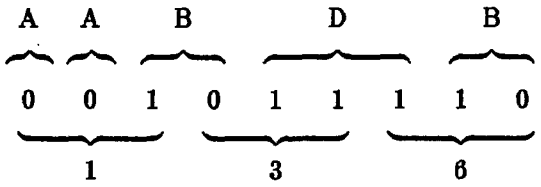
 ACM 0-89791-159-8/85/006/0122 $00.75

tables to isolate each bit of the input into a corresponding byte at the beginning of the decoding process, applying then the usual procedure of the Huffman algorithm. This process can hardly be qualified, however, as efficient. Another approach would be to identify the first coded characters of the first byte of the encoded text, using an appropriate table-lookup, shifting then the non-decoded remaining bits to align them with the first bits of the following byte, and repeating the process until the input is exhausted. In order to avoid bit-shift operations, we can simulate a left (right) shift by repeated accesses to a table that contains the value $2i$ ($\frac{i}{2}$, resp.) at entry i.

Contrary to these two methods, which are merely techniques to circumvent the direct reference to bits in the program, the method to be described below in section 2 aims at a natural and efficient byte-per-byte processing of the input. The decoding subroutine is extremely simple; some preprocessing, however, is required for building a number of tables that should be resident in internal memory during the decoding process. In section 3 some ideas are presented that can reduce the number and size of the required tables, so as to fit the necessary information in a small part of a microcomputer's memory. By using generalized Huffman codes with radix $r > 2$, an approach is presented in section 4, which can again reduce the amount of required internal memory space, at the cost however of loosing some of the optimal performance of binary Huffman codes. Finally experimental results, obtained by applying these different variants on large corpora of English and Hebrew texts, are presented in section 5.

2. The basic idea: Partial-Decoding Tables

Given an alphabet L of N characters, and a message ($\equiv$ sequence of elements of L) to be compressed, we begin by compressing it using the variable-length Huffman codes — hereafter denoted by H-codes — of the different characters of L as computed by the conventional Huffman algorithm. We now partition the resulting bit-string into k-bit blocks, where k is chosen so as to make the processing of k-bit blocks, with the particular machine and high-level language at hand, easy and natural. Clearly, the boundaries of these blocks do not necessarily coincide with those of the H-codes: a k-bit block may contain several H-codes, and an H-code may be split in two (or more) adjacent k-bit blocks. As an example, let $L = \{$A,B,C,D$\}$, with H-codes $\{0, 10, 110, 111\}$ respectively, and choose $k = 3$. Consider the following input string, its coding and the coding's partition into 3-bit blocks:



The last line gives the integer value $0 \leq i < 2^3$ of the block.

Decoding under these conditions is made possible by using a set of m auxiliary tables, which, for a given Huffman code, are prepared in advance in the preprocessing stage. The number of entries in each table is 2^k, corresponding to the 2^k possible values of the k-bit patterns. Each entry is of the form (W, j), where W is a sequence of characters and j ($0 \leq j < m$) is the index of the next table to be used. The idea is that entry i, $0 \leq i < 2^k$, of table number 0 contains, first, the longest possible decoded sequence W of characters from the k-bit block representing the integer i (W may be empty when there are H-codes of more than k bits); usually some of the last bits of the block will not be decipherable, being the prefix P of more than one H-code; j will then be the index of the table corresponding to that prefix (if P is the empty string Λ, $j = 0$). Table number j is constructed in a similar way except for the fact that entry i will contain the analysis of the bit pattern formed by the prefixing of P to the binary representation of i. We thus need a table for every possible proper prefix of the given H-code; the number of these prefixes is obviously equal to the number of internal nodes of the appropriate Huffman-tree (the root corresponding to the empty string and the leaves corresponding to the H-codes), so that $m = N - 1$.

More formally, let P_j, $0 \leq j < N - 1$, be an enumeration of all the proper prefixes of the H-codes (no special relationship needs to exist between j and P_j, except for the fact that $P_0 = \Lambda$). In table j corresponding to P_j, the i-th entry, $T(j, i)$, is defined as follows: let B be the bit-string composed of the juxtaposition

of P_j to the k-bit binary representation of i. Let W be the (possibly empty) longest sequence of characters that can be decoded from B, and P_l the remaining undecipherable bits of B; then $T(j,i) = (W,l)$.

Referring again to the simple example given above, there are 3 possible proper prefixes: Λ,1,11, hence 3 corresponding tables indexed 0,1,2 resp., and these are given in Figure 1. The column headed 'Pattern' contains for every entry the binary string which is decoded in Table 0; the binary strings which are decoded by Tables 1 and 2 are obtained by prefixing '1', resp. '11', to the strings in 'Pattern'.

Entry	Pattern	Table 0		Table 1		Table 2	
	for Table 0	W	l	W	l	W	l
0	000	AAA	0	BAA	0	CAA	0
1	001	AA	1	BA	1	CA	1
2	010	AB	0	BB	0	CB	0
3	011	A	2	B	2	C	2
4	100	BA	0	CA	0	DAA	0
5	101	B	1	C	1	DA	1
6	110	C	0	DA	0	DB	0
7	111	D	0	D	1	D	2

Figure 1: *Partial decoding tables*

For the input example given above, we first access table 0 at entry 1, which yields the output string AA; table 1 is then used with entry 3, giving the output B; finally table 2 at entry 6 gives output DB.

The utterly simple decoding subroutine (for the general case) is as follows (where $S(i)$ denotes the i-th block of the input stream and j is the index of the currently used table):

```
Basic Decoding Algorithm
i ← 1
j ← 0
repeat
    (output, j) ← T(j, S(i))
    print output
    i ← i + 1
until input is exhausted
```

As mentioned before, the choice of k is largely governed by the machine-word structure and the high-level language architecture. A natural choice in most cases would be $k = 8$, corresponding to a byte context, but $k = 4$ (half-byte) or $k = 16$ (half-word) are also conceivable. The larger is k, the greater is the number of characters that can be decoded in a single iteration, thus transferring a substantial part of the decoding time to the preprocessing stage. The size of the tables however grows exponentially with k, and with every entry occupying (for $k = 8$) 2 to 9 bytes, each table may require 1–2K bytes of internal memory. With an alphabet of 20 to 30 characters, these storage requirements may be prohibitive. We now develop an approach that can help reducing the number of required tables and their size.

3. Binary Forests Reducing the Number of Tables

The storage space needed by the partial decoding tables can be reduced by relaxing somewhat the approach of the previous section, and using the conventional Huffman decoding algorithm no more than once for every block, still processing only k-bit blocks. This is done by redefining the tables and adding new data-structures.

Let us suppose, just for a moment, that after deciphering a given block B of the input that contains a "remainder" P (which is a prefix of a certain H-code), we are somehow able to determine the correct suffix of P and its length l, and accordingly its corresponding encoded character. More precisely, since an H-code can extend into more than two blocks, l will be the length of the suffix of P in the next k-bit block which contains also other H-codes, hence $0 \leq l < k$. In the next iteration (decoding of the next k-bit block which was not yet entirely deciphered), table number l will be used, which is similar to table 0, but *ignores* the first l bits of the corresponding entry, instead of *prefixing* P to this entry as in section 2.

Therefore the number of tables reduces from $N - 1$ (about 30 in a typical natural-language case) to only k (8 in a typical byte context), where entry i in table l, $0 \leq l < k$, contains the decoding of the $k - l$ rightmost bits of the binary representation of i. It is clear, however, that table 1 contains two exactly equal halves, and in general table l ($0 \leq l < k$) consists of 2^l identical parts. Retaining then in each table only the first 2^{k-l} entries, we are able to compress the needed k tables into the size of only 2 tables. The entries of the tables are again of

the form (W, j); note however that j is not an index to the next table, but an identifier of the remainder P; it is only after finding the correct suffix of P and its length l that we can access the right table l.

For the same example as before one obtains the tables of figure 2, where table t decodes the bit-strings given in 'Pattern', but ignoring the t leftmost bits, $t = 0, 1, 2$, and $l = 0, 1, 2$ corresponds respectively to the proper prefixes $\Lambda, 1, 11$.

Entry	Pattern	Table 0		Table 1		Table 2	
	for Table 0	W	l	W	l	W	l
0	000	AAA	0	AA	0	A	0
1	001	AA	1	A	1	-	1
2	010	AB	0	B	0		
3	011	A	2	-	2		
4	100	BA	0				
5	101	B	1				
6	110	C	0				
7	111	D	0				

Figure 2: *The Sub-String Translate Tables*

The algorithm will be completed if we can find a method to identify the H-code corresponding to the remainder of a given input block, using of course the following input block(s). We introduce the method through an example.

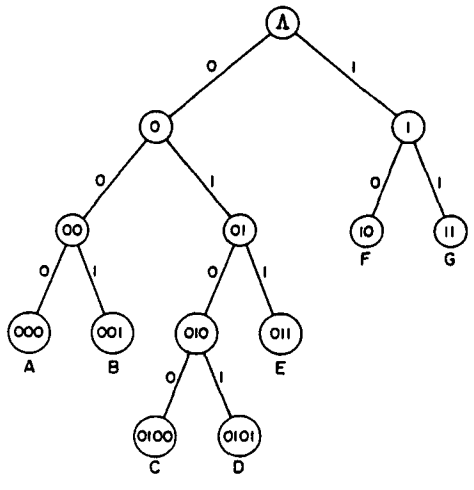


Figure 3: *The Huffman Tree H*

Figure 3 shows a typical Huffman tree for an alphabet L of $N = 7$ characters. For every internal node, the edge pointing to the left son is labelled 0, the one to the right son is labelled 1; the root contains Λ; every other node v contains a binary string which is obtained by concatenating the labels of the edges on the path from the root to v. The strings in the leaves are the Huffman codes of the characters in L.

Assume now $k = 8$ and consider the following adjacent blocks of input: 00101101 00101101. The first block is decoded into the string BE and the remainder $P = 01$. Starting at the internal node containing 01 and following the first bits of the following block, we get the H-code C, and length $l = 2$ for the suffix of P, so that table 2 will be used when decoding the next block; ignoring the first 2 bits, this table translates the binary string 101101.

For the general case, let us for simplicity first assume that the depth of H, which is the length of the longest H-code, is bounded by k. Given the non-empty remainder P of the current input block, we must access the internal node corresponding to P, proceed downwards turning left (0) or right (1) as indicated by the first few bits of the next k-bit block, until we reach a leaf. This leaf contains the next character of the output. The number of levels passed is the index of the table to be used in the next iteration.

Our goal is to simulate this procedure without having to follow a "bit-traversal" of the tree. The algorithm we propose in the sequel uses a binary forest instead of the original Huffman tree H. For the sake of clarity, the construction of the forest is described in two steps.

First, replace H by $N-2$ smaller trees H_i, which are induced by the proper sub-trees rooted at the internal nodes of H, and correspond to all non-empty proper prefixes of the H-codes. The nodes of the forest contain binary strings: Λ for the roots, and for each other node v, similarly to the Huffman tree, a string obtained by concatenating the labels of the edges on the path from the root to v, but padded at the right by zeroes so as to fill a k-bit block. The string in node v is denoted by VAL(v). Figure 4 depicts the forest obtained from the tree of our example, where the pointer to each tree is symbolized by the corresponding proper prefix.

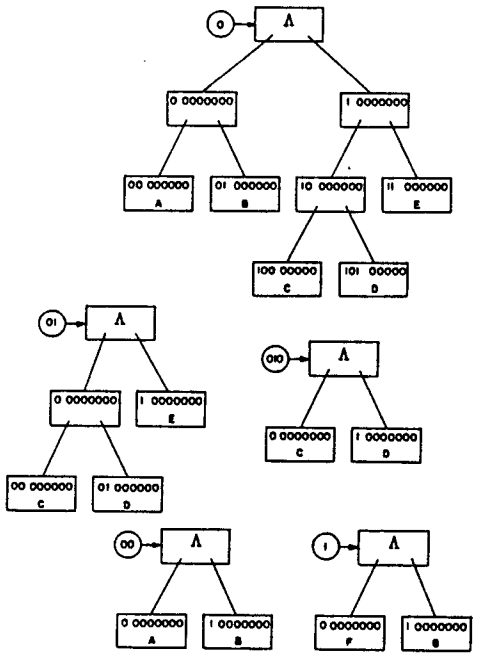


Figure 4: *Forest for Proper Prefixes*

The idea is that the identifier of the remainder in an entry of the tables described above is in fact a pointer to the corresponding tree. The traversal of this tree is guided by the bits of the next k-bit block of the input, which can directly be compared with the contents of the nodes of the tree.

Consider now also the possibility of long H-codes, which extend over several blocks. They correspond to long paths so that the depth of some trees in the forest may exceed k. During the traversal of a tree, passing from one level to the next lowest one is equivalent to advancing one bit in the input string. Hence when the depth exceeds k, all the bits of the current k-bit block were used, and we pass to the next block. Therefore the above definition of VAL(v) applies only to nodes on levels up to k; this definition is generalized to any node by: VAL(v) for a node v on level j, with $ik < j \leq (i+1)k$, $i \geq 0$, is the concatenation of the labels on the edges on the path from level ik to v.

In the second step, we compress the forest as could have been done with any Huffman tree. In such trees, every node has degree 0 or 2, i.e. they appear in pairs of brothers (except the root). For a pair of brother-nodes (a, b), VAL(a) and VAL(b) differ only in the j-th bit, where j is the level of the pair (here and in what follows, the level of the root of a tree is 0), or more precisely, $j = (\text{level} - 1) \pmod k + 1$. In the compressed tree, every pair is represented by an unique node containing the VAL of the right node of the pair, the new root is the node obtained from the only pair in level 1, and the tree structure is induced by the non-compressed tree. Thus a tree of l nodes shrinks now to $(l-1)/2$ nodes. Figure 5 is the compressed form of the forest of figure 4.

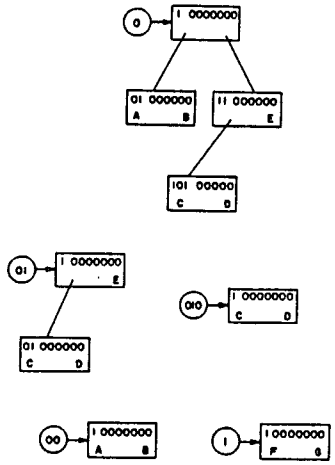


Figure 5: *The Compressed Forest*

After accessing one of the trees, the VAL of its root is compared with the next k-bit block B. If B is smaller, it must start with 0 and we turn left; if B is greater or equal, it must start with 1 and we turn right. This leads to the modified algorithm below. Notations are like before, ROOT(t) points to the t-th tree of the forest, every node has 3 fields: VAL, a k-bit value, LEFT and RIGHT each of which is either a pointer to the next level or contains a character of the alphabet. When accessing table j, the index is taken modulo the size of the table, which is 2^{k-j}.

Revised Decoding Algorithm

```
i ← 1
j ← 0
repeat
    (output, tree-nbr) ← T(j, S(i) mod 2^(k-j))
    print output
    i ← i + 1
    j ← 0
    if tree-nbr ≠ 0 then
        TRAVERSE ( ROOT(tree-nbr) )
until input is exhausted
```

where the procedure TRAVERSE is defined by

```
TRAVERSE ( node )
    repeat
        if S(i) < VAL(node) then
                node ← LEFT(node)
        else
                node ← RIGHT(node)
        if node is a character C then print C
        j ← j + 1
        [[ j is the number of bits
            in S(i) which are 'used up' ]]
        if j = k then
            j ← 0
            i ← i + 1
    until a character was printed
end
```

To evaluate the number of comparisons in the tree, let us suppose that the letters of the alphabet L appear with probabilities p_i, and that the corresponding H-codes c_i have length l_i, $1 \leq i \leq N$. The weighted average length is denoted by WAL $= \sum p_i l_i$. One may assume that the probability of H-code c_i being the last in a k-bit block is proportional to $p_i l_i$, and that for a given H-code, each bit-position has equal chance to be the last in the block. Hence for each k-bit block, the average number of comparisons, ANC, in the forest (the average length of the suffix of the H-code which was split by the "border" of the k-bit block) is given by

$$\mathrm{ANC} = \frac{1}{\mathrm{WAL}} \sum_i p_i l_i \frac{1}{l_i} \sum_{j=1}^{l_i} (l_i - j)$$
$$= \frac{1}{2}\left(\frac{\sum p_i l_i^2}{\mathrm{WAL}} - 1 \right).$$

An obvious upper bound for ANC is $\frac{1}{2}(\max\{l_i\} - 1)$, but generally ANC is not much larger than $\frac{1}{2}(\mathrm{WAL} - 1)$ (see section 5 for numerical examples).

Any node v of the original (compressed) Huffman tree H' generates several nodes in the forest, the number of which is equal to the level of v in H'. Hence the space needed to store the forest depends on the form of H', thus on the frequency distribution. The extreme cases are:

a. A degenerated tree with, e.g., none of the LEFT-fields serving as a pointer; such a tree is obtained by a super-increasing sequence of frequencies f_i, with $f_{i+1} \geq \sum_{j=1}^{i} f_j$, $i = 1, \ldots, N-1$. The corresponding forest has exactly one tree with i nodes for $1 \leq i \leq N-2$, so that the total number of nodes is $O(N^2)$.

b. A full binary tree, obtained by an alphabet of $N = 2^t$ characters all of which appear with equal probability. There are 2^j nodes on level j, $0 \leq j < t$, in the compressed Huffman tree, thus the number of nodes in the forest is $\sum_{j=1}^{t-1} j 2^j < N \log_2 N$.

Therefore the space needed is between $O(N \log N)$ and $O(N^2)$ which is reasonable in most practical applications.

If one agrees to abandon the optimality of the Huffman tree, it is possible to keep the space of the forest bounded by $O(N \log N)$, even for the worst distribution. This can be done by imposing a maximal length of $K = O(\log N)$ to the codewords. If K does not exceed the block-size k, the decoding algorithm can even be slightly simplified, since in the procedure TRAVERSE there is no need to check if the end of the block was reached. An other advantage of bounding the depth of the Huffman tree is that this tends to lengthen the shortest H-code. Since the number of characters stored at each entry in the partial-decoding tables is up to $1 + \lfloor (k-1)/s \rfloor$, where s is the length of the shortest H-code, this can reduce the space required to store each table. An algorithm for the construction of an optimal tree with bounded depth can for example be found in [3].

4. Huffman Codes with Radix $r > 2$

The number of tables can also be reduced by the following simple variants which yield lower compression factors than the methods described above. Let us apply the Huffman algorithm with radix r, $r > 2$, the details of which can be found in Huffman's original paper [1]. In the corresponding r-ary tree, every internal node has r sons, except perhaps one on the next-to-lowest level of the tree which has between 2 and r sons. If we choose $r = 2^l$, we can encode the alphabet in a first stage using r different symbols; then every symbol is replaced by a binary code of l bits. If in addition l divides k, the "borders" of the k-bit blocks never split any l-bit code. Hence in the partial-decoding tables, the possible remainders are sequences of one or more r-ary

symbols. There is therefore again a correspondence between the possible remainders and the internal nodes of the r-ary Huffman tree, only that their number now decreased to $\lceil (n-1)/(r-1) \rceil$. Moreover, there may be some savings in the space needed for a specific table. As we saw before, the space for each table depends on the length s of the shortest H-code, so this can be k with the binary algorithm when $s = 1$, but at most $\lceil k/2 \rceil$ in the 4-ary case.

Due to the restrictions on the choice of r, there are only few possible values. For example, for $k = 8$, one could use a quaternary code ($r = 2^2$), where every code-word has an even number of bits and the number of tables is reduced by a factor of 3, or a hexadecimal code ($r = 2^4$), where the code-word length is a multiple of 4 and the number of tables is divided by 15.

Referring to the Huffman tree given in figure 3, suppose that a letter corresponding to a leaf on level l appears with probability 2^{-l}, then the corresponding 2^2-ary tree is given in figure 6. Note that the only proper prefixes of even length are Λ and 00, so that the number of tables dropped from 6 to 2.

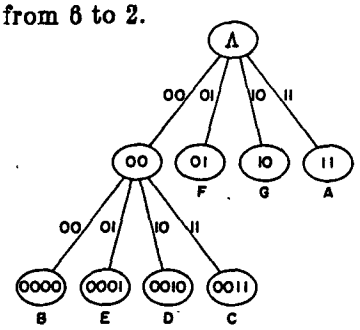


Figure 6: *The Quaternary Huffman Tree*

However, with increasing r, compression will get worse, so that the right trade-off must be chosen according to the desired application.

5. Examples and Experiments

The performance of the proposed methods was evaluated for four different applications, two theoretical "extreme-case" distributions and two real-life cases dealing with natural-language databases:

		Standard Huffman Tree	Partial Decoding Tables	With Binary Forest	Bounded Code Length	Quaternary Code	Hexadecimal Code
DGT	average length	2.000	2.000	2.000	2.266	2.286	4.000
	time units / letter	2.000	0.250	0.5	0.584	0.285	0.5
	nbr of tables	–	31	2	2	11	3
	tree-nodes	31	–	465	173	–	–
	total nbr of bytes	62	71424	5491	4615	14080	2304
FBT	average length	5	5	5	5	5.375	6.25
	time units / letter	5	0.625	1.667	1.667	0.672	0.781
	nbr of tables	–	31	2	2	11	3
	tree-nodes	31	–	98	98	–	–
	total nbr of bytes	62	23040	1314	1314	8448	2304
English	average length	4.185	4.185	4.185	4.191	4.285	4.459
	time units / letter	4.185	0.523	1.285	1.286	0.536	0.557
	nbr of tables	–	25	2	2	9	2
	tree-nodes	25	–	93	89	–	–
	total nbr of bytes	50	25600	2251	2239	11520	1536
Hebrew	average length	4.285	4.285	4.285	4.289	4.353	4.624
	time units / letter	4.285	0.536	1.438	1.438	0.544	0.578
	nbr of tables	–	29	2	2	10	2
	tree-nodes	29	–	112	111	–	–
	total nbr of bytes	58	37120	2556	2553	12800	1536

Table 1: *Experimental results*

1. A degenerated tree (DGT) obtained from an alphabet of 32 letters which appear with probabilities $p_i = 2^{-i}$ for $1 \leq i \leq 31$ and $p_{32} = 2^{-31}$.
2. A full binary tree (FBT) obtained from an alphabet of 32 letters with uniform distribution $p_i = 2^{-5}$ for $1 \leq i \leq 32$.
3. The distribution of the 26 characters in an English text of 100,000 words chosen from many different sources, as given by Heaps [4] (pp. 199–200).
4. The distribution of 30 Hebrew letters (including apostrophes and blank) as computed from the data base of the Responsa Retrieval Project (see for example [5]) of about 40 million Hebrew and Aramaic words.

Six of the methods detailed above were applied for these cases. First the standard Huffman Code (with compressed tree) was tested. The maximal length of an H-code turned out to be 31 for DGT, 5 for FBT, 10 for English and 9 for Hebrew. Choosing $k = 8$, the k-bit blocks approach was then applied, first with the partial-decoding tables (with $N - 1$ tables), and then with the binary forest variant. Obviously the three methods give, by definition, the same optimal compression ratio. Three more methods were now tested, which give reduction in storage requirements at the cost of a lower compression ratio: the bounded code-length approach (limiting the length of H-codes to 8), the quaternary Huffman code and the hexadecimal one.

The results are presented in Table 1. For each variant and each example, the first line gives the average length of an H-code (WAL) in bits.

In order to compare the time-complexity of the different variants, we count both an access to a table and a comparison in a tree as one time-unit. The lines headed "time-units / letter" give the average number of units needed to decipher a single character. This is obviously equal to WAL for the standard Huffman algorithm, and to WAL/k for the simple approach with partial-decoding tables. For the variant with the binary forest the following argument is used: some bits at the beginning of each k-bit block are decoded using one of the trees in the forest, their number being at the average ANC $= \frac{1}{2}\left(\left(\sum p_i l_i^2\right)/\text{WAL} - 1\right)$ (see section 3). A single table access then translates the rest of the block, except again the last ANC bits on the average (the "remainder"), hence the average number of letters obtained by this table access is $(k - 2\,\text{ANC})/\text{WAL}$. Therefore the average number of time units needed for the processing of one character is

$$\frac{\text{ANC} + 1}{(k - 2\,\text{ANC})/\text{WAL} + 1} = \frac{\text{WAL}\,(\text{ANC} + 1)}{k - 2\,\text{ANC} + \text{WAL}}.$$

On our examples, the values of ANC are 1.000 for DGT, 2 for FBT, 1.699 for English and 1.868 for Hebrew.

The entries entitled "tree-nodes" refer to the number of nodes in the (compressed) trees stored in addition to the tables. The total number of bytes required to store the trees and/or tables (last line for each of the examples in Table 1) is calculated as follows. For the standard Huffman tree one needs only 2 bytes per node, for the nodes of the forest, 3 bytes per node. The entries of the partial-decoding tables are chosen large enough for the longest possible sequence of characters (one byte for each), plus one byte for the pointer to the next table or tree. For example, the shortest H-code for English has 3 bits, so that at most 3 characters must be stored in any entry, thus the total space needed by the simple approach of the partial-decoding tables is $(3 + 1) \times 2^8 \times 25 = 25600$ bytes.

Summarizing the results, the different methods should be compared with respect to four parameters: their compression ratio, the ease of programming, the processing time and the additional internal space. The standard Huffman algorithm is bit-oriented, whereas all the variants refer only to bytes and are thus not only simpler, but also considerably faster. On the other hand, the standard algorithm needs practically no internal memory. For the variants, there is a natural trade-off between simplicity and time on one side and space on the other. The fastest method is that with the partial-decoding tables, but for example for English, one needs $25K$ of internal memory. The approach with the forest is 2.5 times slower, but 90% of storage can be saved. At the cost of loosing the optimality of the binary Huffman code by passing to r-ary encoding with $r > 2$, the storage requirements can be reduced drastically for only a small increase in processing time.

REFERENCES

[1] **Huffman D.,** *A method for the construction of minimum redundancy codes,* Proc. of the IRE 40 (1952) 1098–1101

[2] **Even S.,** *Graph Algorithms,* Computer Science Press, 1979.

[3] **Itai A.,** *Optimal Alphabetic Trees,* SIAM J. of Comp. 5 (1976) 9–18.

[4] **Heaps P.,** *Information Retrieval, Computational and Theoretical Aspects,* Academic Press, 1978

[5] **Choueka Y.,** *Full text systems and Research in the Humanities,* Computers and the Humanities XIV (1980) 153–169.