Bidirectional Huffman Coding

A. S. FRAENKEL* AND S. T. KLEIN†
* Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel
† Center for Information and Language Studies, University of Chicago, 1100 East 57-th Street, Chicago, IL 60637, USA

Under what conditions can Huffman codes be efficiently decoded in both directions? The usual decoding procedure works also for backward decoding only if the code has the affix property, i.e., both prefix and suffix properties. Some affix Huffman codes are exhibited, and necessary conditions for the existence of such codes are given. An algorithm is presented which, for a given set of codeword lengths, constructs an affix code, if there exists one. Since for many distributions there is no affix code giving the same compression as the Huffman code, a new algorithm for backward decoding of non-affix Huffman codes is presented, and its worst case complexity is proved to be linear in the length of the encoded text.

Received August 1989

1. INTRODUCTION

For a given sequence of n weights $w_1, \ldots, w_n$, with $w_i > 0$, Huffman's well-known algorithm[9] constructs an *optimum prefix code*. We use throughout the term 'code' as abbreviation for 'set of codewords'. In a prefix code no codeword is the prefix of any other. A Huffman code, consisting of codewords c_i of length l_i, $1 \leq i \leq n$, is optimum in the sense that it minimizes the weighted average length $\Sigma_{i=i}^{n} w_i l_i$. For most sets of weights there is a large number of optimum codes. It is thus natural to look for optimum codes having some *additional* properties. For example Schwartz & Kallick[18] proposed a code for which the codewords, when sorted by their respective weights, are also ordered lexicographically. More recently, Ferguson & Rabinowitz[3] studied Huffman codes having a *synchronizing* codeword c, i.e., if c appears in the encoded string, the codewords following it are recognized, regardless of possible errors preceding c.

In this paper, we investigate Huffman codes as to their ability of allowing efficient bidirectional decoding. The prefix-property ensures that an encoded string can instantaneously be deciphered when it is processed from the beginning towards the end (left to right). However, if we want to decode the encoded text *backwards*, this is easily done only if the Huffman code has also the *suffix property* (which is usually not the case), so that no codeword is the prefix or the suffix of any other. A code having both prefix and suffix properties is called an *affix* code (as in Peterson[14]). Bidirectional decoding may be useful in the following applications:

1) *KWIC display*. In full-text information retrieval systems a query consists of one or several keywords, the occurrences of which are to be located. This is done with the help of a 'concordance' which contains for every different word of the database a list of pointers to all its appearances in the text. A convenient way to present the retrieved items to the user is in form of a 'KeyWord In Context' (KWIC) index (see Heaps[8]): typically, a context of k words is displayed for each occurrence, with one of the keywords centred, where k is a fixed or variable integer chosen by the user. Hence starting with the pointer to the keyword, the k words of the KWIC must be collected processing the text both forward and backward. Therefore the text file is usually encoded by some fixed-length code, which can be very wasteful. Affix Huffman codes can thus be used for text compression, permitting the display of an arbitrarily long segment of text preceding any keyword. This question of KWIC display and other possible applications of affix codes, have been communicated to us by Prof. Y. Choueka, and it was one of the motivations for this work.

2) *Retrieval of truncated terms*. The problem is to find the pattern $*X*$ in a given text file, where X is a string and $*$ is a variable-length-don't-care character, that is, the $*$ represent arbitrary, possibly empty, strings, which complete the string X to a word appearing in the text. In order to retrieve the word matched by the pattern $*X*$, we decode forwards and backwards from the location where X was detected up to the nearest blanks. Again this is done easily with fixed-length codes, but with variable-length codes only if they have the affix property.

3) *Use of tapes*. Some information stored on tapes can be processed during the rewind operations. For instance, the PL/1 programming language allows files on magnetic tapes to be accessed in reverse order using the 'BACKWARDS' attribute. If such a feature is desired, we need affix codes to enable the use of variable-length codes for compression purposes.

4) *Implementation of deques*. Many algorithms use linear lists such as deques which are often implemented by sequential allocation (see Knuth,[11] pp. 240–251). The size of these lists can be reduced when we allow their records to be encoded by a variable-length code, which must have the affix property since records can be retrieved from both ends of a deque.

It should be noted that one can easily construct affix variable-length codes, if one does not insist on optimum compression. For example, fix any binary pattern π and consider the set Y of strings of the form $y = \pi x \pi$, where x is a binary string such that the pattern π appears in y only as prefix and suffix. Clearly, Y is affix, but bidirectional decoding is still possible even if π appears only once, say as prefix, in each codeword as proposed by Gilbert,[5] who was concerned with synchronization.

We shall restrict our discussion to complete finite affix codes, the construction of which is more difficult. A

code $\mathcal{C}$ is *complete* if by adjoining any codeword $c \notin \mathcal{C}$, one obtains a set $\mathcal{C}' = \mathcal{C} \cup \{c\}$ which is not uniquely decipherable. Note that in an application to data communication, it is not always recommended to use a complete affix code, because for such a code (unless it is the trivial code consisting of {0, 1} only) it is well-known that if a bit gets lost or an extraneous bit is picked up, the decoding process will never resynchronize after the error. On the other hand, the possibility for efficient backward decoding may be useful in case of a transmission error. For example, if the original text is in some natural language, the error can be located approximately as the point from which on the decoded text seems to be garbled. The lost tail can be restored at least partially by backward decoding of the compressed text, starting at the end of the message so that the damage of the error can be restricted to a local perturbation.

Affix codes are already mentioned in Gilbert & Moore[6] (where they are called *never-self-synchronizing*) and in Schützenberger;[16] they are extensively studied in Berstel & Perrin[1] [Chapter III] under the name of *biprefix* codes. We further restrict attention to binary codes, but all the results are easily extendable to k-ary codes for any $k \geq 2$. New results on the existence of complete affix codes are presented in the next section. In Section 3 we sketch an algorithm for the construction of a complete affix code, if it exists, and report on experiments which suggest that weight-distributions admitting an affix code giving optimum compression, are rare. Therefore we deal in Section 4 with a method for efficient backward decoding of Huffman codes which lack the suffix property.

A binary code has the property of *finite decipherability delay* if there exists a constant K such that the knowledge of the first K bits of any encoded text $\mathcal{T}$ suffices to determine the first codeword of $\mathcal{T}$. This permits the construction of a decoding automaton with finite memory. For example, every prefix code has finite delay with K = maximum codeword length. The problem with a non-affix Huffman code is that the corresponding reverse code (which is actually used for the backward decoding) does not have finite decipherability delay. This is already mentioned in Levenshtein[12] and proved for every complete prefix code in Schützenberger.[17] It follows that for every non-affix Huffman code, an encoded text can be found, the backward decoding of which recognizes its first codeword only after the entire string is read. In our application for backward decoding, we are however interested in the *average* number of bits which need to be processed before a codeword is recognized. We define accordingly the average decipherability delay of the code, which also corresponds to the average space complexity of the decoding algorithm. We show below that assuming a reasonable probability model, we get that the average decipherability delay is bounded by a constant depending only on the Huffman code.

The reversal of a Huffman code being still a uniquely decipherable (UD) code, one could use known algorithms for decoding general UD codes. For example, Even[2] [Problems 4.1 and 4.2] proposes to scan the entire message in both directions, which for any prefix code as in the Huffman case discussed here, reduces to the simple forward decoding. The new algorithm of Section 4, however, as well as its analysis, assume not only the code being UD, but rely on the fact that it is the reversal of a Huffman code, which yields the constant average delay.

2. EXISTENCE OF AFFIX CODES

There is a natural correspondence between Huffman codes and Huffman trees, and we shall henceforth use the languages of codes (codewords and their lengths) and trees (leaves and their levels) interchangeably. A binary Huffman tree for n weights has $n-1$ internal nodes, from each of which emanate two edges, one labelled '0' and the other labelled '1'. There are thus 2^{n-1} possible Huffman trees, which correspond to 2^{n-1} different assignments of codewords c_i to the weights w_i, for $1 \leq i \leq n$. The number of optimum *codes* is smaller, since by switching the labels of the edges emanating from an internal node, the left and right subtrees of which are identical in structure, the set of codewords is not altered, only their assignment to the weights is permuted. However our search for affix codes should not be restricted to this set of Huffman codes, since there are optimum codes which cannot be obtained via the Huffman algorithm. Consider for example the following sequence of nine weights: $(w_1, \ldots, w_9) = (1, 1, 1, 1, 3, 3, 3, 3, 7)$. The only vector of lengths minimizing $\Sigma w_i l_i$ is $(l_1, \ldots, l_9) = (4, 4, 4, 4, 3, 3, 3, 3, 2)$, but none of the Huffman codes with this length vector has the affix property. On the other hand, the tree depicted in Figure 1(a), which is not a Huffman tree for these weights (since in a Huffman tree, the four leaves with weight 1 belong to the same subtree of depth 2), is nevertheless optimum, and the corresponding code in Figure 1(b) is affix.

We shall therefore consider, for each sequence of weights, the class of all the codes having the same length-vector as the Huffman code for these weights, so that the string $\langle n_1, \ldots, n_l \rangle$, where n_i is the number of codewords of length i, can be used to identify this class. Note that $\Sigma_{i=1}^{l} n_i 2^{-i} = 1$ (see [Ref. 11, Exercise 2.3.4.5–3]). Such a string is called a *quantized source* in Ref. 3, or simply *source* in the sequel. For example the source corresponding to the code in Figure 1 is ⟨0, 1, 4, 4⟩. Most weight distributions have a unique source, but some can have more, for example the sequence of weights (1, 1, 2, 2, 4) has three sources: ⟨1, 1, 1, 2⟩, ⟨1,

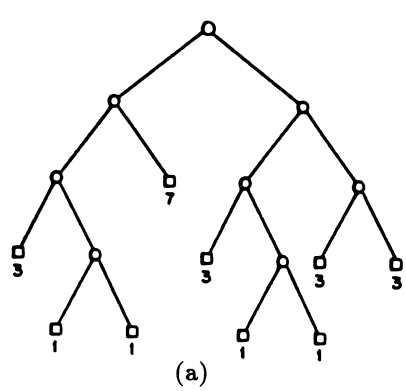

Figure 1. Example of an affix code.

0, 4⟩ and ⟨0, 3, 2⟩. Thus when looking for an affix code for a certain sequence of weights, all its sources must be considered.

Our first concern is to show that our requirements on the suffixes is not too restrictive and that affix codes actually exist. Indeed, every fixed-length code has the affix property, but from the point of view of compression capability, fixed length codes are not very interesting since they are optimum only for uniform or almost uniform distributions. Henceforth any fixed-length code is called a *trivial* affix code.

Proposition 1. There are infinitely many nontrivial affix codes.

Proof: The example in Figure 1 shows that at least one such code exists. The proof is completed if we show that for every given affix code, there is another affix code with a larger number of codewords. Let $A = \{\alpha_1, \ldots, \alpha_n\}$ be an affix code. Consider the set $B = \{\beta_1, \ldots, \beta_{2n}\}$ defined by $\beta_{2i} = \alpha_i 0$, $\beta_{2i-1} = \alpha_i 1$ for $1 \le i \le n$, i.e., the tree corresponding to B is obtained from the tree corresponding to A by splitting all its leaves. Obviously, B is an affix code. ■

On the other hand, there are are sources for which no complete affix code exists.

Proposition 2. Let $S = \langle n_1, \ldots, n_l \rangle$ be a source with $n_1 \neq 0$. Then no nontrivial affix code exists for S.

Proof: If the code is nontrivial, then $l > 1$. From $n_1 \neq 0$ follows that there is precisely one codeword of length 1, which can be either '0' or '1'. But in every complete code, both '0' and '1' are suffixes of certain codewords. ■

Proposition 2 does not apply to commonly encountered distributions, since the existence of a codeword of length 1 implies that one of the weights is at least $(\Sigma w_i)/3$ (Johnsen[10]). The shortest codewords for many distributions of alphabets of natural languages are of length 2 or 3. Let m denote, here and in the sequel, the length of the shortest codeword(s) for a given source, i.e., $m = \min\{i | n_i > 0\}$. Proposition 2 dealt with the case $m = 1$, and in the next theorem we generalize it to larger m. We first introduce the following.

Definition: A *Shift-Register Sequence with word-length* t, denoted in the sequel by $\mathcal{S}_t$, is a sequence of k codewords $\{c_0, \ldots, c_{k-1}\}$, all of length t bits, such that for $0 \le i < k$, the suffix of length $t-1$ of c_i is identical with the prefix of length $t-1$ of $c_{(i+1 \bmod k)}$.

The name of such a 'cyclic sequence' of codewords was chosen because the elements of every $\mathcal{S}_t$ can be viewed as the consecutive states of a shift register of t stages (see for example Golomb[7]). Every $\mathcal{S}_t$ containing k elements is generated by a string of k bits $A = a_0 \cdots a_{k-1}$ which are to be thought of as being written on a circle; the codewords of $\mathcal{S}_t$ are obtained by collecting t consecutive bits of this cyclic string, starting from a_0, and shfiting the starting point for each subsequent codeword by one bit, i.e., $c_i = a_i a_{i\oplus 1} \cdots a_{i\oplus(t-1)}$ for $0 \le i < k$, where, here and below, $\oplus$ denotes addition modulo k. Let $\mathcal{S}_t(A)$ denote $\mathcal{S}_t$ generated by A. For example, the string $A = 010011$ generates $\mathcal{S}_4(A) =$ {0100, 1001, 0011, 0110, 1101, 1010}. The generating string can also be smaller than the word-length, for example $\mathcal{S}_3(10) = \{101, 010\}$.

Theorem 1. A complete affix code is trivial if and only if the set of shortest codewords includes some shift register sequence with word-length m.

Proof: If all the codewords have the same length m bits, then all the 2^m possible bit-patterns are codewords, in particular the string of m zeros, which is the unique element of $\mathcal{S}_m(0)$.

Conversely, suppose that for a given complete affix code $\mathcal{C}$, the set of shortest codewords includes the k elements of some $\mathcal{S}_m(A)$, for $A = a_0 \cdots a_{k-1}$. Let B denote the infinite binary string obtained by concatenating A with infinitely many copies of itself, and let $B(i)$ denote the prefix of length i bits of B. We show by induction on r the truth of the statement $\Gamma(r)$: *'the* 2^r *binary strings defined by* $d_1 \cdots d_r B(m-r)$, *where the bits* d_i *take all their possible values, are also codewords'*.

For $r = 1$, consider the infinite binary string $D = d_1 B$. The string D traces a path through a hypothetical infinite binary tree (with, say, 0 indicating a left turn and 1 a right turn). Exactly one prefix e of D corresponds to a path from the root to a leaf in the finite tree associated with $\mathcal{C}$, since $\mathcal{C}$ is complete. In other words, e is a codeword. If e has more than m bits, then its last m bits are of the form $a_i a_{i\oplus 1} \cdots a_{i\oplus(m-1)}$ for some $0 \le i < k$. By hypothesis, $a_i a_{i\oplus 1} \cdots a_{i\oplus(m-1)}$ is a codeword c_i of $\mathcal{S}_m(A)$. But it is also a suffix of e, contradicting the fact that $\mathcal{C}$ is an affix code. Hence e has exactly m bits so that both $0B(m-1)$ and $1B(m-1)$ belong to $\mathcal{C}$, which proves $\Gamma(1)$.

Suppose the truth of $\Gamma(j)$ for $1 \le j \le r-1 < m$, and consider the infinite binary string $D = d_1 \cdots d_r B$. As above, there is a unique prefix e of D which is a codeword. The length of e is less than $m + r$, since otherwise the suffix of length m of e is one of the codewords of $\mathcal{S}_m(A)$. Moreover, the length of e cannot be $m - 1 + j$ for any $2 \le j \le r$, since otherwise e has a suffix of length m of the form $d_j \cdots d_r B(m - r + j - 1)$, but by $\Gamma(r - j + 1)$ this is a codeword. Thus e has exactly m bits which proves $\Gamma(r)$.

Summarizing, $\Gamma(i)$ is true for all i, in particular for $i = m$, which means that $\mathcal{C}$ is a fixed-length code. ■

Note that for $m = 1$ we get Proposition 2, since both {0} and {1} are $\mathcal{S}_1$'s. The importance of Theorem 1 is that it will guide us in the construction of affix codes in the following section. It also gives information on the maximal size of the set of shortest codewords, as will now be shown.

Corollary 1. Let $S = \langle n_1, \ldots, n_l \rangle$ be a source. If $n_m > H(m)$, where $H(m) = 2^m - (1/m) \sum_{i=1}^{m} 2^{(i,m)}$ and (x, y) denotes the greatest common divisor of x and y, then no nontrivial complete affix code exists for S.

Proof: Consider the set A of the 2^m binary strings of length m, and define a binary relation $\mathcal{R}$ on A by: elements x and y of A are related by $\mathcal{R}$ if and only if one can be obtained by cyclically shifting the other. Clearly, $\mathcal{R}$ is an equivalence relation, so let $Z(m)$ be the number of equivalence classes in the partition of A

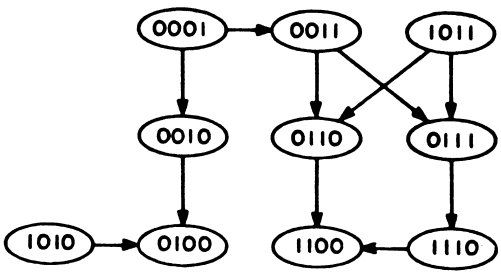

Figure 2. Example of a set of $H(4) = 10$ strings including no $\mathcal{S}_4$.

induced by $\mathcal{R}$. It is easy to see that every equivalence class is an $\mathcal{S}_m$. From the disjointness of the classes follows that at least one element of every class must be excluded from the set of shortest codewords, so that an affix code can exist only if $n_m \leq 2^m - Z(m)$. The corollary thus follows from Golomb[7] [Chapter VI, Theorem 1], where it is shown that $Z(m) = (1/m) \sum_{i=2}^{m} 2^{(i,m)}$. ■

We note in passing that the generalization of Corollary 1 to k-ary codes implies $Z_k(m) = (1/m) \sum_{i=1}^{m} k^{(i,m)}$. In particular, $\sum_{i=1}^{m} k^{(i,m)} \equiv 0 \pmod m$, for all positive integers m and k. See also Riordan[15] [Chapter VI, Problem 37].

Since $H(1) = 0$, we again get Proposition 2 for $m = 1$. For $m = 2$, the corollary states that there can be only one shortest codeword of length 2; for $m = 3$ at most 4 codewords can be of shortest length and for $m = 4$ at most 10.

Note that the corollary does not affirm the existence of a set of codewords of size $H(m)$ not including an $\mathcal{S}_m$. Indeed, the $\mathcal{S}_m$ used in the partition are all generated by strings of length m, where in each $\mathcal{S}_m$ multiple elements are dropped. For example, for $m = 3$, the generating strings can be 000, 001, 011 and 111, yielding respectively the $\mathcal{S}_m$: {000}, {001, 010, 100}, {011, 101, 110} and {111}. However, other $\mathcal{S}_m$ can exist, which are generated by strings of length $\neq m$, for example $\mathcal{S}_3(01) = \{010, 101\}$, which also have to be avoided. Therefore if we want to construct a set of shortest codewords of size $H(m)$, the element to be dropped from each class must be chosen carefully and it is not clear that this can be done so as to avoid *every* shift register sequence. For $m \leq 4$ it *is* possible to choose sets of size $H(m)$, for example:

for $m = 2$: {01}
for $m = 3$: {001,010,110,011}
for $m = 4$ {0001,0011,0111,1110,1100,1011,1010, 0110,0010,0100}.

An easy way to see that a set of codewords contains no $\mathcal{S}_m$ is to consider the following directed graph: the vertices are the 2^m binary strings of length m and there is a directed edge from v to w if w is the successor of v in some shift-register sequence, i.e., the $m - 1$ leftmost bits of w coincide with the $m - 1$ rightmost bits of v. The full graph of 2^m vertices, where each vertex has indegree and outdegree 2, is known as the (binary) de Bruijn diagram of order m (see Ref. 7). A set C of codewords includes no $\mathcal{S}_m$, if and only if the subgraph of the de Bruijn diagram induced by C contains no directed cycles. Figure 2 shows the subgraph corresponding to the set of size $H(4)$ of the above example.

On the other hand, we have no proof that for all possible values of n_m there actually exists an affix code having a set C of n_m shortest codewords, even if C can be chosen so that it includes no $\mathcal{S}_m$. For $m \leq 3$, we found examples of affix codes for all the possible values of n_m, as shown in Fig. 1 and in the four examples of Fig. 3.

Using repeatedly the proof of Proposition 1, we conclude from the last example that there exist affix codes for sources with $m \geq 3$ and $n_m = 2^{m-1}$. Note also that

0 0 1	0 0 1	0 0 1	0 0 1
0 0 0 0	1 0 1	0 1 0	0 1 0
0 1 0 0	0 0 0 0	1 1 0	0 1 1
0 1 0 1	0 1 0 0	0 0 0 0	1 1 0
0 1 1 0	0 1 1 0	0 1 1 1	0 0 0 0
0 1 1 1	0 1 1 1	1 0 0 0	1 0 0 0
1 0 0 0	1 0 0 0	1 0 1 1	1 1 1 1
1 0 1 0	1 1 0 0	1 1 1 1	1 1 1 0 0
1 0 1 1	1 1 1 0	1 1 1 0 0	1 1 1 0 1
1 1 0 0	1 1 1 1	1 1 1 0 1	1 0 1 1 1
1 1 0 1	1 1 0 1 0	1 0 1 0 0	1 0 1 0 0
1 1 1 0	1 1 0 1 1	1 0 1 0 1	1 0 1 0 1
1 1 1 1	1 0 0 1 0	1 0 0 1 1	0 0 0 1 1 1
1 0 0 1 0	1 0 0 1 1	0 1 1 0 0	0 0 0 1 0 0
1 0 0 1 1	0 1 0 1 0	0 1 1 0 1	0 0 0 1 0 1
0 0 0 1 0	0 1 0 1 1	0 0 0 1 1	1 0 0 1 1 1
0 0 0 1 1	0 0 0 1 0	0 0 0 1 0 0	1 0 0 1 0 0
	0 0 0 1 1	0 0 0 1 0 1	1 0 0 1 0 1
		1 0 0 1 0 0	1 0 1 1 0 0
		1 0 0 1 0 1	1 0 1 1 0 1
			1 0 0 1 1 0 0
			1 0 0 1 1 0 1
			0 0 0 1 1 0 0
			0 0 0 1 1 0 1

Figure 3. Examples of affix codes with 1, 2, 3, 4 shortest codewords of 3 bits.

the example for $\langle 0, 0, 2, 8, 8\rangle$ is not the one obtained from the example in Figure 1 via the proof of Proposition 1.

Summarizing, Theorem 1 and its corollary provide necessary, but not sufficient conditions for the existence of a complete affix code.

3. CONSTRUCTION OF AFFIX CODES

Given a source $S = \langle n_1, \ldots, n_l\rangle$, we are interested in an algorithm which constructs an affix code for S if there exists one. The number of complete prefix codes for a given source is

$$\mathcal{N}(S) = \prod_{i=1}^{l} \binom{2^i - \Sigma_{j=1}^{i-1} 2^{i-j} n_j}{n_i},$$

since once the set A of codewords of length $< i$ is fixed, we can choose the n_i codewords of length i among the set of 2^i possible codewords, from which we exclude all those having an element of A as prefix, and there are 2^{i-j} codewords of length i having the same prefix of length j. For certain sources, the number of prefix codes can be large enough to make an exhaustive search for an affix code prohibitive. For example, the source of the distribution of the English alphabet, as given in Ref. 8, is $\langle 0, 0, 2, 7, 7, 5, 1, 1, 1, 2\rangle$, and there are 127,733,760 different prefix codes for this source.

Every Huffman tree has the property that any of its subtrees T_i is a Huffman tree for the weights corresponding to the leaves of T_i. However the affix property is not 'hereditary', in other words it is not true that any subtree of an affix tree has the affix property. For example, the left subtree of Figure 1a is not affix by Proposition 2. Hence a bottom-up construction as the one used by Huffman will not work in our case.

A first cut-back in the number of potential affix codes is obtained from the following theorem which is cited in Ref. 6 and proved in Ref. 16. For a source $S = \langle n_1, \ldots, n_l\rangle$, let

$$d(S) = 1n_1 2^{-1} + 2n_2 2^{-2} + \cdots + ln_l 2^{-l}.$$

The quantity $d(S)$ is called the *degree* of the source S.

Theorem (*Gilbert, Moore, Schützenberger*). The degree of any complete affix code is an integer.

For example, the tree in Figure 1a has degree 3 and all the trees in Figure 3 have degree 4. The above source for English yields a non-integer degree, so there is no affix code for the English alphabet giving the same compression as the Huffman code.

The converse of the above theorem is not true, i.e., there are sources with integral degree, but for which no complete affix code exists. For example the source $\langle 1, 0, 4\rangle$ has degree 2 but satisfies the conditions of Proposition 2.

The algorithm for the construction of an affix code for a given source S therefore starts by evaluating $d(S)$, and proceeds only if this is an integer. The tree is then constructed top-down, level by level. There are N_i nodes (internal or leaves) on level i, where

$$n_i = \begin{cases} n_l & \text{for } i = l \\ n_i + N_{i+1}/2 & \text{for } i < l. \end{cases}$$

On level i, n_i among the N_i nodes must be chosen to be leaves in the final tree, the remaining nodes on level i are split, giving the $2(N_i - n_i) = N_{i+1}$ nodes of level $i + 1$. The procedure is most easily understood considering for a given source S the following 'tree of trees' $\mathcal{T}(S)$, which we introduce through the example in Figure 4 for $S = \langle 1, 0, 2, 4\rangle$, $l = 4$ and $(N_1, \ldots, N_4) = (2, 2, 4, 4)$.

Every node on level i of $\mathcal{T}(S)$, $0 \le i \le l$, contains a binary tree of depth i which corresponds to the upper part of one of the possible trees for the source S, and every tree in a node is an extension of the tree in the father-node. In particular, the root of $\mathcal{T}(S)$ contains a

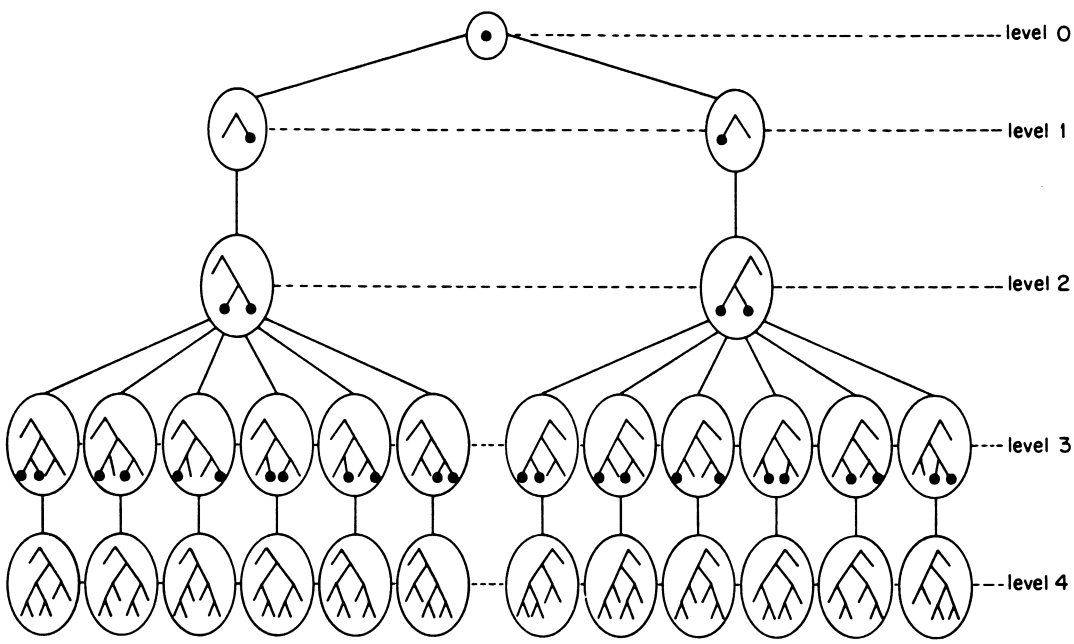

Figure 4. The 'tree of trees' $\mathcal{T}(\langle 1, 0, 2, 4\rangle)$.

binary tree of depth zero, which is a unique node. The tree in a node on level i of $\mathcal{T}(S)$ has N_i leaves on its level i, $N_i - n_i$ of which are marked by a dot as being internal nodes in the final tree. Every node on level $i < l$ of $\mathcal{T}(S)$ has $\binom{N_{i+1}}{n_{i+1}}$ sons, which all contain the same binary tree and differ only in the choice of the dotted leaves. The leaves of $\mathcal{T}(S)$ contain all the possible binary trees for the source S, so there are $\mathcal{N}(S)$ leaves, in our example $\mathcal{N}(\langle 1, 0, 2, 4\rangle) = 12$.

A preorder traversal of the tree $\mathcal{T}(S)$ may be used to generate all the possible trees for S in an exhaustive search: every time a leaf is reached, the corresponding tree is checked for the affix property. We can however do better, checking already in higher levels of $\mathcal{T}(S)$ if there are affix-trees in its nodes. There is no need to visit a subtree rooted at a node of $\mathcal{T}(S)$ which contains a non-affix tree. If on the other hand, it contains an affix tree, the number of sons of this node which should be visited will often be much smaller than $\binom{N_i}{n_i}$, as we consider only extensions which preserve the affix property. Therefore large parts of $\mathcal{T}(S)$ can be pruned so that the time complexity of traversing the tree is reduced. In the worst case it will still be equivalent to exhaustive search, but for almost all our experiments we got on-line results in a few seconds.

Description of the algorithm

The algorithm for searching for an affix tree performs a recursive preorder traversal of the tree $\mathcal{T}(S)$, parts of which are dynamically pruned. It stops with a positive answer when one of the leaves is reached, or with a negative answer at the end of the traversal.

To **initialize,** we start with the full binary tree with m levels (level m of $\mathcal{T}(S)$). There are $\binom{2^m}{n_m}$ possibilities to choose the leaves, and there is one node in $\mathcal{T}(S)$ for each of the choices, but because of Theorem 1, only sets of n_m codewords not including an $\mathcal{S}_m$ need to be considered. The number of sets to be checked can be reduced further, by remarking that for every given affix code the set of the binary complements of the codewords is also an affix code; hence it suffices to consider only sets of n_m codewords such that the leftmost bit of at least $\lceil n_m/2 \rceil$ of them is zero, and even this set can be restricted further: for example for $m = 3$ and $n_m = 2$, there is no need to check both $\{001, 101\}$ and $\{110, 010\}$. The general step is now called for each node x on level m of $\mathcal{T}(S)$, such that the tree in x satisfies these conditions.

The **general step** for level $i > m$ is called with a node x on level $i - 1$ of $\mathcal{T}(S)$ as parameter. The dotted leaves of the tree in x are split, giving a tree T_x with N_i nodes on level i, which are partitioned into two disjoint subsets: INT – the set of those which must be internal nodes because they correspond to one of the 'forbidden' codewords, i.e., they have another codeword (corresponding to a leaf on level $j < i$ of T_x) as suffix; and LV – the set of those nodes which can be leaves.

If $|\mathrm{LV}| < n_i$, then the tree T_x cannot be extended to an affix tree for S, thus the subtree of $\mathcal{T}(S)$ rooted at x can be pruned, and we return from this recursive call. Otherwise T_x can be extended, but we need only to consider a subset of the sons of x. Indeed, if $i < l$, we choose n_i among the $|\mathrm{LV}|$ 'permitted' leaves on level i in T_x, so the number of sons of x in $\mathcal{T}(S)$ which need to be considered is only $\binom{|\mathrm{LV}|}{n_i}$ instead of $\binom{N_i}{n_i}$. The general step is then recursively repeated with each of these sons as parameter. The subtrees of $\mathcal{T}(S)$ rooted at the other sons are pruned.

If $i = l$ (note that $\binom{N_l}{n_l} = 1$), we have reached a leaf of $\mathcal{T}(S)$ and the tree in this leaf is affix, so we are done.

An exact analysis of this algorithm seems not to be an easy task. The problem is to evaluate the size of the set LV for every internal node of $\mathcal{T}(S)$. If we bound $|\mathrm{LV}|$ by its possible maximum (N_i for a node on level i of $\mathcal{T}(S)$), we get $O(n^2\mathcal{N}(S))$, where $n = \Sigma_{i=i}^{l} n_i$, as bound for the complexity of the algorithm, which is not polynomial. However, an intuitive argument could be that the deeper we get into the tree $\mathcal{T}(S)$, the more codewords are fixed for every node and the smaller LV will be, since it must avoid a larger set of suffixes. Thus even if there is a large number of alternatives to be checked on the first few levels, the branching on the next levels will be much more restricted, and deeper levels will often not be reached.

It is not even easy to check the validity of this argument experimentally, say on the sources of various 'real-life' distributions, because sources with integral degree are not abundant. For the weight distributions of the characters of seven natural languages, we have constructed Huffman codes and codes which are optimum subject to the additional constraint that the lengths of the codewords are bounded by some integer D, for various values of D smaller than the depth of the Huffman tree.[4] None of the 48 sources we got has an integral degree. For a 26-letter alphabet, there are 40115 possible sources $\langle n_1, \ldots, n_l\rangle$ with $l \leq 11$; only 77 of them have integral degrees, and from these, 47 are rejected by Proposition 2 or Corollary 1. We have applied the algorithm on the remaining sources and found that $\langle 0, 1, 1, 3, 9, 8, 4\rangle$ is the only source with $l \leq 11$ and $\Sigma_{i=1}^{l} n_i = 26$ for which there exists a complete affix code. The algorithm was generally fast with the number of calls to the 'general step', on these examples of up to 32 characters, rarely exceeding 150. The largest example on which we have tried the algorithm was for the source $S = \langle 0, 0, 2, 0, 10, 24, 8\rangle$, for which we knew by the proof of Proposition 1 that there exists an affix code, since we found one for $\langle 0, 1, 0, 5, 12, 4\rangle$. The algorithm had to visit 139,293 nodes of $\mathcal{T}(S)$, but compared to the possible maximum, this is still only $1.2 \times 10^{-7}\mathcal{N}(S)$.

Since apparently there are many 'real-life' distributions which have no affix code, we treat in the next section the problem of backward decoding of (not necessarily affix) Huffman codes.

4. BACKWARD DECODING OF HUFFMAN ENCODED STRINGS

We are given a finite alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ the elements of which are called letters, and a text $T \in \Sigma^*$ we wish to encode using some binary code $C = \{c_1, \ldots, c_n\}$ of the letters, i.e., if the text is $T = \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k}$ then the corresponding encoded text is $\mathcal{C} = c_{i_1}c_{i_2}\cdots c_{i_k}$, with $i_j \in \{1, \ldots, n\}$ for $1 \leq j \leq k$. For such Σ and C, we define the *associated binary tree* $\mathcal{A}(\Sigma, C)$ as follows: every edge in $\mathcal{A}(\Sigma, C)$ pointing to a left (resp. right) son is labelled 0 (resp. 1); to each element of C corresponds a unique node of $\mathcal{A}(\Sigma, C)$ such that

the binary string c_i is obtained by concatenating the labels of the edges on the path from the root to the node corresponding to the codeword c_i; $\mathcal{A}(\Sigma, C)$ is the union of all these paths, in other words, every node corresponds to a prefix of at least one of the codewords; the node corresponding to c_i is labelled σ_i.

If a Huffman code $H = \{h_1, \ldots, h_n\}$ is used, the associated tree $\mathcal{A}(\Sigma, H)$ is called a *Huffman tree* and has the following properties:

i: since every Huffman code is prefix, there are no internal nodes which are labelled, hence a node is labelled if and only if it is a leaf;
ii: since every Huffman code is complete, all the internal nodes have two sons.

Due to property **i**, the standard decoding procedure is very simple: the algorithm uses a pointer P to the current position in the Huffman tree; P points initially to the root. In the general step, the algorithm inspects the next bit b of the encoded text $\mathcal{C}$ and updates P to point to the left (resp. right) son of the current position, if $b = 0$ (resp. if $b = 1$). If now P points to a leaf, its label σ (which is the corresponding element of Σ) is printed, and P is reset to the root.

Property **ii**, which is a consequence of the optimality of Huffman codes, implies that *any* binary string can be 'decoded'. This can be a disadvantage, as an error in the input string may only be detected at its end, if at all.

For the backward decoding algorithm, we define $\bar{H} = \{\bar{h}_1, \ldots, \bar{h}_n\}$ as the set of reversed Huffman codewords, and we shall use the associated tree $\mathcal{A}(\Sigma, \bar{H})$. If H is an affix code, then $\bar{H}$ is prefix so that $\mathcal{A}(\Sigma, \bar{H})$ is a Huffman tree and the standard decoding procedure can be used. If H is not affix, property **i** does not hold for $\bar{H}$ and we must at each stage of the backward processing of $\mathcal{C}$ keep track of all the possible decodings up to this point. Somewhat surprisingly, the absence of property **ii** will then help to resolve these local ambiguities.

For example, let $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ and $H = \{0, 100, 101, 11\}$. The trees $\mathcal{A}(\Sigma, H)$ and $\mathcal{A}(\Sigma, \bar{H})$ are depicted in Figure 5, in which labelled nodes are represented by squares with their labels written above, and non-labelled nodes are circles. In $\mathcal{A}(\Sigma, \bar{H})$ the nodes are numbered to facilitate references. Note for $\mathcal{A}(\Sigma, \bar{H})$ the lack of property **i** (note 2 is labelled although it is not a leaf) and property **ii** (nodes 2, 4 and 5 are internal with only one son).

The backward decoding algorithm maintains a linear linked list $\mathcal{L}$, the elements E of which are of the form $E = (P, S)$, where P is one of the nodes of the tree $\mathcal{A}(\Sigma, \bar{H})$ and $S \in \Sigma^*$ is a, possibly empty, string of letters

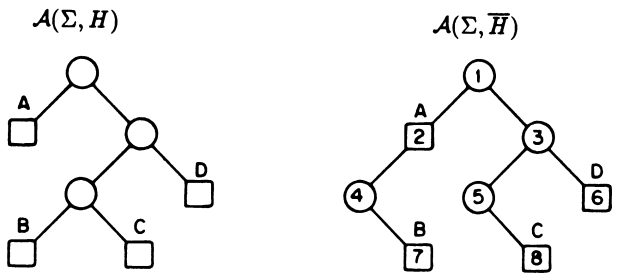

Figure 5. Associated trees for $H = \{0, 11, 100, 101\}$.

of the alphabet. Every element E represents one of the possible decodings of a given suffix of the encoded text $\mathcal{C}$, with P being the current node in the tree and S the current string of decoded letters. For each bit read from the encoded text, there is one iteration during which the list $\mathcal{L}$ is updated, so that at the end of the iteration $\mathcal{L}$ will contain *all* the possible decodings up to this point. Using the above example, suppose a suffix of $\mathcal{C}$ is 011100, which should be read from right to left. Then the list at the end of the 6-th iteration is $\mathcal{L} = \{(5, \mathtt{DAA}), (2, \mathtt{DB}), (1, \mathtt{ADB})\}$.

Description of the algorithm

Prelude: to initialize, the list contains a single element (P, S), with P being the root of $\mathcal{A}(\Sigma, \bar{H})$ and S the empty string Λ. Now the encoded text is processed starting with the rightmost bit and going left.

After having read the next bit b, the **main iteration** is entered. Here and below, an *iteration* of the algorithm is the processing of one bit of the encoded text. The list $\mathcal{L}$ is scanned linearly and each of its elements $E = (P_E, S_E)$ is updated according to b in the following way. Let Q be the left (resp. right) son of P_E if $b = 0$ (resp. $b = 1$). If P_E does not have such a son (i.e., if Q is now the null-pointer **nil**), then the element E cannot be the decoding of the suffix of $\mathcal{C}$ processed so far, so we delete E from $\mathcal{L}$. Suppose now that the son Q of P_E exists. If Q is a leaf labelled, say, by σ_i, this means that for the present decoding E a new codeword was detected, thus P_E is reset to the root and σ_i is concatenated to the left of S_E. Hence suppose that Q is an internal node. Then there is no need to change S_E, only P_E is set to Q. However, if Q is labelled, say, by σ_j, a new element $F = (\text{root}, S_F)$ must be adjoined to $\mathcal{L}$, where S_F is obtained by concatenating σ_j to the left of S_E. This will happen if the last bit(s) read from $\mathcal{C}$ are ambiguous in the sense that they can form a codeword (this possibility is accounted for by the decoding F) and also the proper suffix of one or more other codewords (represented by the updated form of E). The element F should be adjoined to $\mathcal{L}$ in such a way that it will not be processed any more in the present iteration.

To continue the previous example, suppose that the next bit read (the 7-th from the end) is $b = 0$. Scanning the elements of $\mathcal{L}$, $(5, \mathtt{DAA})$ is deleted because node 5 has no left son; $(2, \mathtt{DB})$ is transformed into $(4, \mathtt{DB})$, since the left son of 2 exists but is not labelled; $(1, \mathtt{ADB})$ is transformed into $(2, \mathtt{ADB})$, but since 2 is a node labelled by A, a new element $(1, \mathtt{AADB})$ is adjoined to the list so that finally $\mathcal{L} = \{(4, \mathtt{DB}), (1, \mathtt{AADB}), (2, \mathtt{ADB})\}$.

Let $\mathcal{S} = \{S_E : E = (P_E, S_E) \in \mathcal{L}\}$ denote the set of strings in the elements of $\mathcal{L}$. Let SUF be the longest suffix which is common to all the elements of $\mathcal{S}$ (in the example, SUF = DB). Since $\mathcal{L}$ contains all the possible decodings, the bits corresponding to SUF are unambiguous, therefore SUF can be transferred to the output buffer. Then SUF is deleted from the right end of each element in $\mathcal{S}$, yielding in our example $\mathcal{L} = \{(4, \Lambda), (1, \mathtt{AA}), (2, \mathtt{A})\}$. We refer to the updated strings in $\mathcal{S}$ as the *truncated* strings S_E. In fact, the algorithm would be valid even without these truncations, which are only needed to bound the space complexity. This terminates the iteration for the current bit b.

Postlude: after having exhausted the input string, the

algorithm scans the list $\mathcal{L}$ until an element $E = (P_E, S_E)$ is found for which P_E = root (we show below that there is always exactly one such element). The corresponding string S_E is transferred to the output buffer and this completes the decoding of $\mathcal{C}$. If in the example above, the seventh bit is the last one, i.e., $\mathcal{C} = 0011100$, then the string AA is transferred to the output buffer which therefore contains AADB; this is the desired decoding of $\mathcal{C}$.

Using the fact that property **ii** does not hold, it is easy to implement an error-detecting mechanism. If at some iteration the list $\mathcal{L}$ is completely emptied or if at the end there is no element with P_E = root, this means that there is no possible decoding, thus an error must have occurred. If an error has transformed one codeword into another of equal size, this cannot be detected. In the other case however, chances are good to get stuck after a small number of iterations.

Analysis

The complexity of the algorithm obviously depends on $|\mathcal{L}|$, the size of the list. The list grows by 1 every time a labelled internal node is reached and shrinks by 1 every time the **nil**-pointer is encountered, thus $|\mathcal{L}|$ depends on the structure of the tree $\mathcal{A}(\Sigma, \bar{H})$. In the worst case, the complexity could a priori be exponential: if in some iteration, all the elements of $\mathcal{L}$ point to labelled internal nodes of $\mathcal{A}(\Sigma, \bar{H})$, the size of $\mathcal{L}$ doubles in this iteration. We show that this cannot happen and moreover that the complexity is linear in the length of the decoded text.

Theorem 2. At the beginning of each iteration, distinct elements of $\mathcal{L}$ point to internal nodes of $\mathcal{A}(\Sigma, \bar{H})$ which are on different levels.

Proof: For a node P, denote by $c(P)$ the binary string obtained by concatenating the labels of the edges on the (reversed) path from P to the root, i.e., $c(P)$ is a codeword h_i of the Huffman code H if P is a labelled (square) node, or it is the suffix of one of the codewords h_i if P is an unlabelled internal node. Let b_j be the bit processed in the j-th iteration. Consider two elements $E = (P_E, S_E)$ and $F = (P_F, S_F)$ of $\mathcal{L}$ at the beginning of the i-th iteration, that is after having processed the suffix $B = b_{i-1} \cdots b_1$ of the encoded text.

Since all the elements of $\mathcal{L}$ are obtained by processing some prefix of the string B, it follows that either $c(P_E)$ is a prefix of $c(P_F)$, or $c(P_F)$ is a prefix of $c(P_E)$. Thus if P_E and P_F are on the same level l (which is the length in bits of $c(P_E)$), we must have $c(P_E) = c(P_F)$. On the other hand, both S_E and S_F are decodings of the binary string $b_{i-l-1} \cdots b_1$. This implies $S_E = S_F$ because the Huffman code is uniquely decipherable. Hence if P_E and P_F are on the same level, then $E = F$.

It remains to show that $\mathcal{L}$ never contains multiple copies of any element E. Suppose this assertion is not true and let j be the index of the first iteration which contains at its beginning at least two copies of some element $E = (P_E, S_E)$. Since in the first iteration $\mathcal{L}$ contains only one element, we have $j > 1$. If P_E is not the root, then there were at least two copies of (Q, S_E) at the beginning of iteration $j - 1$, where Q is the father of P_E in $\mathcal{A}(\Sigma, \bar{H})$. Hence we may suppose P_E = root. Let $S_E = \sigma'_s \cdots \sigma'_1$, where $\sigma'_i \in \Sigma$, and let l be the length of the codeword h'_s corresonding to σ'_s in the Huffman code. Then $h'_s = b_{j-1} \cdots b_{j-l}$, and at the beginning of iteration $j - 1$, there were at least two copies of the element $(Q, \sigma'_{s-1} \cdots \sigma'_1)$ in $\mathcal{L}$, where Q is the node such that $c(Q) = b_{j-2} \cdots b_{j-l}$. Thus for every possibility of P_E, we get a contradiction to the minimality of j. ■

Corollary 2. The postlude of the algorithm is well-defined.

Proof: From Theorem 2 follows that after having processed the input string, there is at most one element in $\mathcal{L}$ pointing to the root. On the other hand there is at least one, since $\mathcal{L}$ contains the true decoding. ■

Corollary 3. The worst case time complexity of the backward decoding algorithm for an encoded text of k bits is $O(k)$.

Proof: By Theorem 2, the number of elements in $\mathcal{L}$ can never exceed the number of levels in $\mathcal{A}(\Sigma, \bar{H})$, which is bounded by $|\Sigma| - 1$. In every iteration there is a constant amount of work to be done for the updating of each element of $\mathcal{L}$. In order to find the longest common suffix SUF, the rightmost letters of all the strings S_E are compared. If they are all identical (call this a successful comparison), this letter is transferred to the output buffer and the strings are updated. Then the process is repeated until the first unsuccessful comparison, i.e., until not all the rightmost letters of the strings S_E are identical. There are thus possibly several successful comparisons, followed by a single unsuccessful one for every iteration. If the input string has k bits, the total number of unsuccessful comparisons is exactly k. Although the number of successful comparisons for a given iteration may reach $O(k)$, the total number of successful comparisons in the k iterations is the number of letters in the output string, which is clearly bounded by k. Summarizing, the time complexity of the algorithm is $O(|\Sigma|k)$, but since the size of the alphabet is a constant not depending on k, this is $O(k)$. ■

Corollary 4. The number of elements in $\mathcal{L}$ increases at most by 1 per iteration.

Proof: By Theorem 2, as every new added element points to the root. ■

The example in Figure 5 shows that the number of elements in the list $\mathcal{L}$ can actually reach the number of levels of $\mathcal{A}(\Sigma, \bar{H})$. This, however, is not true for every Huffman code. As we saw in the proof of Theorem 2, $c(P_E)$ is a prefix of $c(P_F)$ or conversely, for all P_E and P_F in $\mathcal{L}$. It follows that at the beginning of each iteration, all the strings $c(P_E)$ for P_E in $\mathcal{L}$ are prefixes of the same binary string B, which is the longest among the strings $c(P_E)$. On the other hand, we know that there are no two strings $c(P_E)$ of equal length. Therefore the number of elements in $\mathcal{L}$ can be l, where l is the number of levels in $\mathcal{A}(\Sigma, \bar{H})$, only if B is a suffix of length $l - 1$ of one of the longest codewords, and there is one element in $\mathcal{L}$ for every prefix of B. An example showing that the maximal size of $\mathcal{L}$ may be less than the depth of

$\mathcal{A}(\Sigma, \bar{H})$ is the Huffman code $H = \{00, 01, 100, 110, 111, 10100, 10101, 10110, 10111\}$. The depth of the tree is 5. However, there are no nodes in $\mathcal{A}(\Sigma, \bar{H})$ corresponding to the strings 010 and 011, but the suffix of length $l-1$ of every longest codeword has one of these two strings as prefix. Hence $|\mathcal{L}|$ cannot exceed 4 in this example. The exact bound on the size of the list can therefore be refined to:

$$\max_i\{\text{number of prefixes } p \text{ of a proper suffix of } h_i: \exists P \in \mathcal{A}(\Sigma, \bar{H}) \quad \text{with } p = c(P)\}. \tag{2}$$

Note that this bound can be evaluated using only the Huffman code since it does not depend on the encoded text.

For the average time complexity, one could imagine that if the chances to reach a labelled internal node are much larger than the chances of being at a node having only one son and proceeding in the direction of the missing son, then $\mathcal{L}$ would have a tendency to grow constantly up to its maximal possible size. This would force us to choose the Huffman code so as to minimize this bias. Fortunately one can show that in a certain sense, the tree $\mathcal{A}(\Sigma, \bar{H})$ is 'balanced' for every Huffman code H, as will be shown in the next theorem.

Let $\mathcal{C} = \mathcal{C}(\Sigma, \bar{H})$ denote the binary tree which is obtained from $\mathcal{A}(\Sigma, \bar{H})$ by adding the missing son to all the internal nodes which have only one son. The added nodes are called **nil**-nodes. Using $\mathcal{C}$ instead of $\mathcal{A}(\Sigma, \bar{H})$, a part of the algorithm can be reformulated as follows: *'for a given bit of input and element of $\mathcal{L}$, proceed from the current node in the direction indicated by the bit; if the new current node is a* **nil***-node, discard this element of $\mathcal{L}$'*. We define the *position* of an element $E = (P_E, S_E) \in \mathcal{L}$ in a given iteration as the node $P_E \in \mathcal{C}$ which is reached after having proceeded, but before discarding the element from $\mathcal{L}$ (if P_E is a **nil**-node) or resetting it to the root (if P_E is labelled). Hence the position of an element of $\mathcal{L}$ can be any node of $\mathcal{C}$, except the root.

In order to evaluate the average time complexity, we assume a probability model in which the probability for an arbitrary element of $\mathcal{L}$ to have its position on a node on level i of $\mathcal{C}$ is proportional to 2^{-i}, for $i > 0$. This model corresponds to a *dyadic* probability distribution over the alphabet, i.e., the probability of occurrence of every letter $\sigma \in \Sigma$ is an integral power of 2^{-1}. There cannot be too great a difference between the actual probability distribution and the dyadic one assumed in the model, since both yield the same Huffman tree. In Longo & Galasso,[13] the set of probability distributions over a finite alphabet is given a 'pseudometric', and an upper bound is derived for the distance from any probability distribution to the dyadic distribution giving the same Huffman tree.

Denote by N the increase in the size of $\mathcal{L}$ caused by the processing of an arbitrary element E of $\mathcal{L}$, i.e., N is a random variable which assumes values from $\{-1, 0, 1\}$.

Theorem 3. For the given probability model, the expected value of N is zero.

Proof: The expectation of N, $\mathsf{E}(N)$, is evaluated by conditioning on the position of the given element in the tree $\mathcal{C}$:

$$\mathsf{E}(N) = \sum_{v \in \mathcal{C}\setminus\{\text{root}\}} \mathsf{E}(N|\text{position is } v)\mathsf{P}(\text{position is } v) \tag{3}$$

Let $l(v)$ denote the level of v in $\mathcal{C}$ and $K = \Sigma_{w \in \mathcal{C}\setminus\{\text{root}\}} 2^{-l(w)}$. Then we have in our model

$$\mathsf{P}(\text{position is } v) = \frac{1}{K} 2^{-l(v)}. \tag{4}$$

Let U, V and W respectively denote the set of labelled internal nodes, the set of **nil**-nodes and the set of labelled leaves of $\mathcal{C}$. Once the position of the element of $\mathcal{L}$ is fixed, the value of N is determined, so

$$\mathsf{E}(N|\text{position is } v) = \begin{cases} 1 & \text{if } v \in U; \\ -1 & \text{if } v \in V; \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Thus it follows from (3)–(5) that

$$\mathsf{E}(N) = \frac{1}{K}\left(\sum_{v \in U} 2^{-l(v)} - \sum_{v \in V} 2^{-l(v)}\right). \tag{6}$$

But each labelled node is at the same level as in the Huffman tree, which is complete, so that

$$\sum_{v \in U} 2^{-l(v)} + \sum_{v \in W} 2^{-l(v)} = 1. \tag{7}$$

On the other hand, the leaves $V \cup W$ of $\mathcal{C}$ also constitute a complete code, thus

$$\sum_{v \in V} 2^{-l(v)} + \sum_{v \in W} 2^{-l(v)} = 1. \tag{8}$$

From (6)–(8) follows $\mathsf{E}(N) = 0$. ■

Theorem 3 should be understood as a property of the tree $\mathcal{C}(\Sigma, \bar{H})$, namely that the number and position of the labelled internal nodes and the **nil**-nodes are closely related, independently of the code H. We cannot infer from Theorem 3 that the expected size of $\mathcal{L}$ will be constant, because we assumed an ideal probability model. For instance, during the processing of the first i bits of the encoded text, the probability for an element of $\mathcal{L}$ to have its position on level $j > i$ is zero. However our experiments, which are described below, indicate that the model is quite close to what happens in 'real-life' examples, and we actually found that the size of $\mathcal{L}$ is more or less constant, rather than only bounded as implied by Corollary 3.

The space complexity of the algorithm is defined to be the total lengths of the truncated strings S_E which are stored simultaneously in $\mathcal{L}$. Since the time complexity is $O(k)$, also the space complexity is $O(k)$ for an encoded text of length k bits and because of the infinite decipherability delay of non-affix Huffman codes, the worst case space complexity is $\Omega(k)$. For example, using again the Huffman code of Figure 5, suppose that the text is $\mathtt{D}'\mathtt{B}$ (where x^r denotes the concatenation of r copies of the string x), yielding the encoded text $1^{2r+1}00$. Then the list at the end of iteration $2j+1$, for $1 \le j \le r+1$, is $\mathcal{L} = \{(3, \mathtt{D}^{j-1}\mathtt{AA}), (1, \mathtt{D}^{j-1}\mathtt{B})\}$, and the strings cannot be truncated.

There is always exactly one element in $\mathcal{L}$ cor-

responding to the true decoding of the current part of the encoded text. The other elements are called *false elements*. For every false element E, let IT(E) be the index of the first iteration for which the decoding corresponding to E differs from the true decoding. In other words, IT(E) is either the index of the iteration at which E was created, or it is the index of the iteration in which E 'gave birth' to a new element F, which corresponds to the true decoding, so E with its updated pointer is a false element. To evaluate the average space complexity, weaker probability assumptions than in Theorem 3 are sufficient.

Theorem 4. Suppose there is a real constant $q > 0$ such that q is a lower bound on the probability of every false element E to be discarded in any iteration with index $i >$ IT(E). Then the average space complexity, and thus the average decipherability delay, is $O(1)$.

Remark: Under the assumption of the model and the notations of Theorem 3, we get

$$q = (1/K)\,\Sigma_{w \in \{\text{nil-nodes}\}}\, 2^{-l(w)}.$$

Proof: For every i we define the index $t(i) < i$ such that at the beginning of iteration i, the string which was already transferred to the output buffer is the decoding of $b_{t(i)} \cdots b_1$. Hence, all the truncated strings S_E in the elements of $\mathcal{L}$ at the beginning of iteration i are decodings of some suffixes of the string $B_i = b_i b_{i-1} \cdots b_{t(i)+1}$. Consider for iteration i the false element E^i such that IT(E^i) is minimal among all the IT(E) for $E \in \mathcal{L}$ (by Corollary 4, exactly one such element exists for each i). Then IT(E^i) $\leq t(i) + 1$, since otherwise one or more of the rightmost bits of B_i are unambiguous and their decoding could have been transferred to the output buffer. The lengths of the truncated strings S_F for $F \in \mathcal{L}$, which are clearly bounded above by the length of the string B_i, are therefore bounded above by $i -$ IT(E^i). But in our model, for any false element E, the number of iterations from IT(E) until E is deleted is a geometrically distributed random variable with probability of success $\geq q$. Therefore the expected value of $i -$ IT(E^i) is bounded above by $1/q$. This bound, as well as the bound on the number of elements in $\mathcal{L}$, depending only on the Huffman code, not on the encoded text, the expected space complexity is $O(1)$. ■

Experimental Results

We have applied the backward decoding algorithm on various texts and collected the following statistics. The first text was an English technical text of 77000 uppercase characters not containing any special symbols, except blank. In order to check the algorithm on a different natural language, we chose as second text the 98681 Hebrew characters of the book of Genesis (including one blank after each word). Finally we wanted to check the influence of the size of the alphabet, so we took as third text a technical paper of 90000 characters which was used as input file to Knuth's TEX typesetting system. The texts were Huffman coded and the associated trees of reversed codewords were constructed.

Table 1 lists some statistical information on the three files: the size of the alphabet Σ, the source obtained from Huffman's algorithm, the average length of a codeword in bits, the number of nodes in the associated tree of reversed codewords and the possible maximal length of the list $\mathcal{L}$, which was computed using (2). Note that the latter was less than the depth of the tree for all three examples. None of the sources admits an affix code, because they all have non-integral degree.

The backward decoding algorithm was then applied on substrings of various lengths of the encodings of the original texts, and with various starting points. Table 2 summarizes the values which were obtained for the

Table 1. Statistics of Huffman codes

Text	$\lvert\Sigma\rvert$	Source	Average codeword length	Nbr of nodes in $\mathcal{A}(\Sigma, \bar{H})$	Possible maximum $\lvert\mathcal{L}\rvert$
English	27	⟨0, 0, 2, 7, 7, 4, 2, 3, 2⟩	4.16	67	8
Hebrew	28	⟨0, 1, 0, 7, 6, 4,6,4⟩	4.11	67	7
TEX-input	89	⟨0, 0, 1, 6, 7, 8, 11, 6, 12, 19, 6, 6, 2, 3, 2⟩	4.90	338	9

Table 2. Experimental results

Length of encoded text (chars)	English			Hebrew			TEX-input		
	$\lvert\mathcal{L}\rvert$	length of S_E		$\lvert\mathcal{L}\rvert$	length of S_E		$\lvert\mathcal{L}\rvert$	length of S_E	
	average	max	average	average	max	average	average	max	average
1000	5.107	20	6.534	5.018	23	5.020	6.845	26	7.526
10000	5.134	31	6.925	5.009	30	5.118	6.875	47	7.526
30000	5.126	36	6.864	5.033	30	5.079	6.900	57	8.630
50000	5.127	38	6.858	5.041	35	5.134	6.915	64	9.074
70000	5.128	38	6.841	5.045	35	5.137	6.925	69	8.871
90000				5.048	35	5.143	6.925	69	8.805

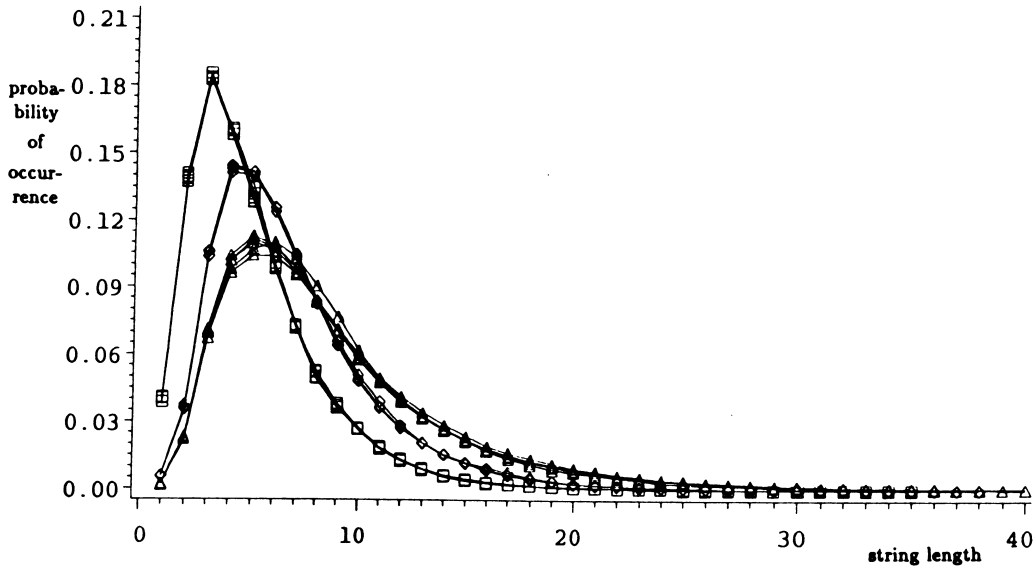

Figure 6. Distribution of lengths of the strings S_E.

number of elements in the list $\mathcal{L}$ at the end of each iteration and for the number of characters in the strings S_E. The maximal possible values for $|\mathcal{L}|$ were obtained for all the examples, and in fact already for encoded texts of 50 characters length.

The average number of elements of $\mathcal{L}$ can be seen to be practically constant. The experiments showed that the three most frequent values ({4, 5, 6} for English and Hebrew and {6, 7, 8} for T_EX-input) occurred about 90%, 96% and 85% of the time respectively, independently of the length of the encoded text.

As to the strings S_E which are stored in the elements of $\mathcal{L}$, their maximal length increases only slowly with the size of the text, and their average length again seems to be constant. The distributions of the lengths of S_E for the experiments with 10000 characters or more are plotted in Figure 6, which gives for each possible length the probability of its occurrence. The points for English are represented by diamonds, those for Hebrew by squares and those of T_EX-input by triangles. For each text, the corresponding graphs are practically overlapping (the values extending beyond the limits of the figure are all smaller than 0.0003). This suggests that the distribution of the lengths of the S_E is a function of the Huffman code only.

The lengths of the strings deserve some special attention. In contrast to the list $\mathcal{L}$, the elements of which are allocated only when they are needed, the straightforward way to store the strings is by reserving in each element of $\mathcal{L}$ enough space for the longest possible string. We thus need some a priori knowledge of the maximal length. On the other hand this approach can be very wasteful. We can circumvent the problem as follows: the maximal size M of the strings will be fixed arbitrarily; if M has to be exceeded, one or more new elements are adjoined to $\mathcal{L}$ immediately following the current one and serving as its 'continuation'. Therefore the algorithm must check during the processing of each element if it has continuation-elements, but this will increase the execution time only by a small constant factor. In practice, the size M can usually be chosen small enough to satisfy the given constraints on available space, and large enough to get a very small probability for having continuation elements.

When constructing a KWIC-index (application 1 of the introduction), the expected length of the strings gives also information about how far we must go backwards from the located keyword. From Fig. 6 we can conclude that if k words *preceding* the keyword are wanted, there is only a very small probability that one must decode more than $k + 2$ words, or about 11 bits more than for the words *following* the keyword.

We have also collected statistics on the number of times the position of the elements of $\mathcal{L}$ were on level i of $\mathcal{C}(\Sigma, \bar{H})$, and found that it was indeed nearly proportional to 2^{-i}, as assumed in the model used in Theorem 3. Another interesting feature was the change in the size of $\mathcal{L}$ for consecutive iterations. We saw already in Corollary 4 that $|\mathcal{L}|$ cannot increase by more than 1. The maximal *decrease* again depends on the structure of the tree $\mathcal{A}(\Sigma, \bar{H})$: it was 3, 3 and 2 for the English, Hebrew and T_EX-input texts respectively.

5. CONCLUDING REMARKS AND FUTURE WORK

The possibility to decode a variable-length encoded text in both directions may lead to savings in various applications in which fixed-length codes were normally used. The gain is not only in space, but often also in time: more information can be read in each input operation, thus reducing the number of needed I/O accesses, and this generally compensates largely for the time spent on decompressing.

Once the source of the optimum Huffman code is given, the first step should be to check if it is possible to build an affix code. If this is not the case, backward decoding is still possible using the algorithm of the previous section, without change in the order of magnitude of the complexity.

There are several open questions which we leave for further research:

(1) How can the results of Section 2 be extended? In

particular can one formulate necessary and sufficient conditions for the existence of affix codes for certain sources?

(2) Is there a polynomial algorithm which, for a given source $\langle n_1, \ldots, n_l \rangle$ of a complete code (i.e., with $\Sigma n_i 2^{-i} = 1$) with integral degree, constructs an affix code, whenever there exists one? Or perhaps can it be shown that even the decision problem whether such an affix code exists is NP-complete?

(3) How can one find the complete affix code giving best compression, if there is one? Perhaps should one not insist on completeness, since non-complete codes though never optimum for compression, enhance error-detection? How can one find the optimum (not necessarily complete) affix code? There is always one, since any fixed length code is affix.

(4) For the backward decoding algorithm, how can one choose the Huffman code so as to minimize the possible maximal length of the list $\mathcal{L}$ and thus the worst case time complexity?

REFERENCES

1. J. Berstel and D. Perrin, *Theory of Codes*. Academic Press Inc., Orlando, Florida (1985).
2. S. Even, *Graph Algorithms*. Computer Science Press (1979).
3. T. J. Ferguson and J. H. Rabinowitz, Self-synchronizing Huffman codes, *IEEE Trans. on Inf. Th.* **IT-30**, 687–693 (1984).
4. A. S. Fraenkel and S. T. Klein, Bounding the depth of search trees, to appear in *The Computer Journal*.
5. E. N. Gilbert, Synchronization of binary messages, *IRE Trans. on Inf. Th.* **IT-6**, 470–477 (1960).
6. E. N. Gilbert and E. F. Moore, Variable-length binary encodings. *The Bell System Technical Journal* **38**, 933–968 (1959).
7. S. W. Golomb, *Shift Register Sequences*. Aegean Park Press, Laguna Hills, California (1982).
8. H. S. Heaps, *Information Retrieval, Computational and Theoretical Aspects*. Academic Press, New York (1978).
9. D. Huffman, A method for the construction of minimum redundancy codes, *Proc. of the IRE* **40**, 1098–1101 (1952).
10. O. Johnsen, On the redundancy of binary Huffman codes. *IEEE Trans. on Inf. Th.* **IT-26**, 220–222 (1980).
11. D. E. Knuth, *The Art of Computer Programming, Vol I, Fundamental algorithms*. Addison-Wesley, Reading, Mass. (1973).
12. V. I. Levenshtein, Certain properties of code systems. *Soviet Physics-Doklady* **6**, 858–860 (1962).
13. G. Longo and G. Galasso, An application of informational divergence to Huffman codes. *IEEE Trans. on Inf. Th.* **IT-28**, 36–43 (1982).
14. J. L. Peterson, Computer programs for detecting and correcting spelling errors. *Comm. ACM* **23**, 676–687 (1980).
15. J. Riordan, *An Introduction to Combinatorial Analysis*. John Wiley & Sons Inc., New York (1958).
16. M. P. Schützenberger, On a special class of recurrent events. *Ann. Math. Stat.* **32**, 1201–1213 (1961).
17. M. P. Schützenberger, On a question concerning certain free monoids. *J. Comb. Theory* **1**, 437–442 (1966).
18. E. S. Schwartz, B. Kallick, Generating a canonical prefix encoding. *Comm. ACM* **7**, 166–169 (1964).