Fast and Compact Prefix Codes⋆

Travis Gagie[1,2], Gonzalo Navarro[2], and Yakov Nekrich[3]

[1] Research Group in Genome Informatics
Bielefeld University
travis.gagie@gmail.com

[2] Department of Computer Science
University of Chile
gnavarro@dcc.uchile.cl

[3] Department of Computer Science
University of Bonn
yasha@cs.uni-bonn.de

Abstract. It is well-known that, given a probability distribution over n characters, in the worst case it takes $\Theta(n \log n)$ bits to store a prefix code with minimum expected codeword length. However, in this paper we first show that, for any ϵ with $0 < \epsilon < 1/2$ and $1/\epsilon = \mathcal{O}(\mathrm{polylog}(n))$, it takes $\mathcal{O}(n \log \log(1/\epsilon))$ bits to store a prefix code with expected codeword length within an additive ϵ of the minimum. We then show that, for any constant $c > 1$, it takes $\mathcal{O}\!\left(n^{1/c} \log n\right)$ bits to store a prefix code with expected codeword length at most c times the minimum. In both cases, our data structures allow us to encode and decode any character in $\mathcal{O}(1)$ time.

1 Introduction

Compression is most important when space is in short supply, so popular compressors are usually heavily engineered to reduce their space usage. Theory has lagged behind practice in this area, however, and there remain basic open questions about the space needed for even the simplest kinds of compression. For example, while compression *with* prefix codes is familiar to any student of information theory, very little has been proven about compression *of* prefix codes. Suppose we are given a probability distribution P over an alphabet of n characters. Until fairly recently, the only general bounds known seem to have been, first, that it takes $\Theta(n \log n)$ bits in the worst case to store a prefix code with minimum expected codeword length and, second, that $\mathcal{O}(n)$ bit suffice to store a prefix code with expected codeword length within 1 of the minimum.

In 1998 Adler and Maggs [1] showed it generally takes more than $(9/40)n^{1/(20c)}$ $\log n$ bits to store a prefix code with expected codeword length at most $cH(P)$,

---

where $H(P)$ is P's entropy and a lower bound on the expected codeword length. (In this paper we consider only binary codes, and by log we always mean $\log_2$.) In 2006 Gagie [6, 7] (see also [8]) showed that, for any constant $c \geq 1$, it takes $\mathcal{O}(n^{1/c} \log n)$ bits to store a prefix code with expected codeword length at most $cH(P) + 2$. He also showed his upper bound is nearly optimal because, for any positive constant ϵ, we cannot always store a prefix code with expected codeword length at most $cH(P) + o(\log n)$ in $\mathcal{O}(n^{1/c-\epsilon})$ bits. Gagie proved his upper bound by describing a data structure that stores a prefix code with the prescribed expected codeword length in the prescribed space and allows us to encode and decode any character in time at most proportional to its codeword's length. This data structure has three obvious defects: when $c = 1$, it is as big as a Huffman tree, whereas its redundancy guarantee can be obtained with just $\mathcal{O}(n)$ bits [10]; when $H(P)$ is small, a possible additive increase of 2 in the expected codeword length may be prohibitive; and it is slower than the state of the art.

In this paper we answer several open questions related to efficient representation of codes. First, in Section 3 we show that, for any ϵ with $0 < \epsilon < 1/2$ and $1/\epsilon = \mathcal{O}(\text{polylog}(n))$, it takes $\mathcal{O}(n \log \log(1/\epsilon))$ bits to store a prefix code with expected codeword length within an additive ϵ of the minimum. Thus, if we can tolerate an additive increase of, say, 0.01 in the expected codeword length, then we can store a prefix code using only $\mathcal{O}(1)$ bits per character. Second, in Section 4 we show that, for any constant $c > 1$, it takes $\mathcal{O}(n^{1/c} \log n)$ bits to store a prefix code with expected codeword length at most c times the minimum, with no extra additive increase. Thus, if we can tolerate a multiplicative increase of, say, 2.01, then we can store a prefix code in $\mathcal{O}(\sqrt{n})$ bits. In both cases, our data structures allow us to encode and decode any character in $\mathcal{O}(1)$ time on a unit-cost word RAM.

2   Related work

A simple pointer-based implementation of a Huffman tree takes $\mathcal{O}(n \log n)$ bits and it is not difficult to show this is an optimal upper bound for storing a prefix code with minimum expected codeword length. For example, suppose we are given a permutation π over n characters. Let P be the probability distribution that assigns probability $1/2^i$ to the $\pi(i)$th character, for $1 \leq i < n$, and probability $1/2^{n-1}$ to the $\pi(n)$th character. Since P is dyadic, every prefix code with minimum expected codeword length assigns a codeword of length i to the $\pi(i)$th character, for $1 \leq i < n$, and a codeword of length $n-1$ to the $\pi(n)$th character. Therefore, given any prefix code with minimum expected codeword length and a bit indicating whether $\pi(n-1) < \pi(n)$, we can find π. Since there are $n!$ choices for π, in the worst case it takes $\Omega(\log n!) = \Omega(n \log n)$ bits to store a prefix code with minimum expected codeword length.

Considering the argument above, it is natural to ask whether the same lower bound holds for probability distributions that are not so skewed, and the answer is 'no'. A prefix code is *canonical* [19] if the first codeword is a string of 0s and any other codeword can be obtained from its predecessor by adding 1, viewing

its predecessor as a binary number, and appending some number of 0s. (See, e.g., [16, 14] for more recent work on canonical codes.) Given any prefix code, without changing the length of the codeword assigned to any character, we can put the code into canonical form by just exchanging left and right siblings in the code-tree. Moreover, we can reassign the codewords such that, if a character is lexicographically the jth with a codeword of length ℓ, then it is assigned the jth consecutive codeword of length ℓ. It is clear that it is sufficient to store the codeword length of each character to be able to reconstruct such a code, and thus the code can be represented in $\mathcal{O}(n \log L)$ bits, where L is the length of the longest codeword.

The above gives us a finer upper bound. For example, Katona and Nemetz [13] showed that, if a character has probability p, then any Huffman code assigns it a codeword of length at most about $\log(1/p)/\log \phi$, where $\phi \approx 1.618$ is the golden ratio, and thus L is at most about $1.44 \log(1/p_{\min})$, where $p_{\min}$ is the smallest probability in P. Alternatively, one can enforce a value for L and pay a price in terms of expected codeword length. Milidiú and Laber [15] showed how, for any $L > \lceil \log n \rceil$, we can build a prefix code with maximum codeword length at most L and expected codeword length within $1/\phi^{L - \lceil \log(n + \lceil \log n \rceil - L) \rceil - 1}$ of the minimum. Their algorithm works by building a Huffman tree T_1; removing all the subtrees rooted at depth greater than L; building a complete binary tree T_2 of height h whose leaves are those removed from T_1; finding the node v at depth $L - h - 1$ in T_1 whose subtree T_3's leaves are labelled by characters with minimum total probability (which they showed is at most $1/\phi^{L - \lceil \log(n + \lceil \log n \rceil - L) \rceil - 1}$); and replacing v by a new node whose subtrees are T_2 and T_3.

A simple upper bound for storing a prefix code with expected codeword length within a constant of the minimum, follows from Gilbert and Moore's proof [10] that we can build an alphabetic prefix code with expected codeword length less than $H(P) + 2$ and, thus, within 2 of the minimum. Moreover, in an optimal alphabetic prefix code, the expected codeword length is within 1 of the minimum [18, 20] which, in turn, is within 1 of the entropy $H(P)$. In an alphabetic prefix code, the lexicographic order of the codewords is the same as that of the characters, so we need store only the code-tree and not the assignment of codewords to characters. If we store the code-tree in a succinct data structure due to Munro and Raman [17], then it takes $\mathcal{O}(n)$ bits and encoding and decoding any character takes time at most proportional to its codeword length. This can be improved to $\mathcal{O}(1)$ by using table lookup, but doing so may worsen the space bound unless we also restrict the maximum codeword length, which may in turn increase the expected codeword length.

The code-tree of a canonical code can be stored in just $\mathcal{O}(L^2)$ bits: By its definition, we can reconstruct the whole canonical tree given only the first codeword of each length. Unfortunately, Gagie's lower bound [7] suggests we generally cannot combine results concerning canonical codes with those concerning alphabetic prefix codes.

Constant-time encoding and decoding using canonical code-trees is simple. Notice that if two codewords have the same length, then the difference between

their ranks in the code is the same as the difference between the codewords themselves, viewed as binary numbers. Suppose we build an $\mathcal{O}(L^2)$-bit array A and a dictionary D supporting predecessor queries, each storing the first codeword of each length. Given the length of a character's codeword and its rank among codewords of the same length (henceforth called its offset), we can find the actual codeword by retrieving the first codeword of that length from A and then, viewing that first codeword as a binary number, adding the offset minus 1. Given a binary string starting with a codeword, we can find that codeword's length and offset by retrieving the string's predecessor in D, which is the first codeword of the same length; truncating the string to the same length in order to obtain the actual codeword; and subtracting the first codeword from the actual codeword, viewing both as binary numbers, to obtain the offset minus 1. (If D supports numeric predecessor queries instead of lexicographic predecessor queries, then we store the first codewords with enough 0s appended to each that they are all the same length, and store their original lengths as auxiliary information.) Assuming it takes $\mathcal{O}(1)$ time to compute the length and offset of any character's codeword given that character's index in the alphabet, encoding any character takes $\mathcal{O}(1)$ time. Assuming it takes $\mathcal{O}(1)$ time to compute any character's index in the alphabet given its codeword's length and offset, decoding takes within a constant factor of the time needed to perform a predecessor query on D. For simplicity, in this paper we consider the number used to represent a character in the machine's memory to be that character's index in the alphabet, so finding the index is the same as finding the character itself.

In a recent paper on adaptive prefix coding, Gagie and Nekrich [9] (see also [12]) pointed out that if $L = \mathcal{O}(w)$, where w is the length of a machine word, then we can implement D as an $\mathcal{O}(w^2)$-bit dictionary data structure due to Fredman and Willard [5] such that predecessor queries take $\mathcal{O}(1)$ time. (We note that Beame and Fich's well-known lower bound [2] on predecessor queries does not apply when the size of the dictionary is proportional to the length of a word.) This seems a reasonable assumption since, for any string of length m with $\log m = \mathcal{O}(w)$, if P is the probability distribution that assigns to each character probability proportional to its frequency in the string, then the smallest positive probability in P is at least $1/m$; therefore, the maximum codeword length in either a Huffman code or a Shannon code for P is $\mathcal{O}(w)$. Gagie and Nekrich used $\mathcal{O}(n \log n)$-bit arrays to compute the length and offset of any character's codeword given that character's index in the alphabet, and vice versa, and thus achieved $\mathcal{O}(1)$ time for both encoding and decoding.

A technique we will use to obtain our first result, presented in section 3, is the *wavelet tree* of Grossi et al. [11], and more precisely the multiary variant due to Ferragina et al. [3]. The latter represents a sequence $S[1, n]$ over an alphabet Σ of size σ such that the following operations can be carried out in $\mathcal{O}\left(\frac{\log \sigma}{\log \log n}\right)$ time on the RAM model with a computer word of length $\Omega(\log n)$: (1) Given i, retrieve $S[i]$; (2) given i and $a \in \Sigma$, compute $rank_a(S, i)$, the number of occurrences of a in $S[1, i]$; (3) given j and $a \in \Sigma$, compute $select_a(S, j)$, the position in S of the j-th occurrence of a. The wavelet tree requires $nH_0(S) + \mathcal{O}\left(\frac{n \log \log n}{\log_\sigma n}\right)$ bits of space,

where $H_0(S) \leq \log \sigma$ is the *empirical zero-order entropy* of S, defined as $H_0(S) = H(\{\mathrm{occ}(a, S)/n\}_{a \in \sigma})$, where $\mathrm{occ}(a, S)$ is the number of occurrences of a in S. Thus $nH_0(S)$ is a lower bound to the output size of any zero-order compressor applied to S. It will be useful to write $H_0(S) = \sum_{a \in \sigma} \frac{\mathrm{occ}(a,S)}{n} \log \frac{n}{\mathrm{occ}(a,S)}$.

Our second result is based on constructing a length-restricted canonical code with maximum codeword length L. We divide all symbols into "probable" symbols that are assigned codewords of length at most $L/c + 2$ and "improbable" symbols that are assigned codewords of length greater than $L/c + 2$. It will be shown in section 4 that all "probable" symbols can be encoded and decoded in $O(1)$ time using $O(n^{1/c} \log n)$ bits. We replace all codewords of length at least $L/c + 3$ with codewords of length L, so that the "improbable" symbols can be encoded and decoded in constant time but we do not have to store the new codewords explicitly.

3 Additive increase in expected codeword length

In this section we exchange a small additive penalty over the optimal prefix code for a space-efficient representation of the encoding, which in addition enables encode/decode operations in constant time.

It follows from Milidiú and Laber's bound [15] that, for any ϵ with $0 < \epsilon < 1/2$, there is always a prefix code with maximum codeword length $L = \lceil \log n \rceil + \lceil \log_\phi(1/\epsilon) \rceil + 1$ and expected codeword length within an additive

$$\frac{1}{\phi^{L - \lceil \log(n + \lceil \log n \rceil - L) \rceil - 1}} \quad \leq \quad \frac{1}{\phi^{L - \lceil \log n \rceil - 1}} \quad \leq \quad \frac{1}{\phi^{\log_\phi(1/\epsilon)}} \quad = \quad \epsilon$$

of the minimum. The techniques described in the previous section give a way to store such a code in $\mathcal{O}(L^2 + n \log L)$ bits, yet it is not immediately obvious how to do constant-time encoding and decoding. Alternatively, we can achieve constant-time encoding and decoding using $\mathcal{O}(w^2 + n \log n)$ bits for the code-tree, if $L = \mathcal{O}(w)$.

To achieve constant encoding and decoding times without ruining the space, we use multiary wavelet trees. We use a canonical code, and sort the characters (i.e., leaves) alphabetically within each depth, as described in the previous section. Let $S[1, n]$ be the sequence of depths in the canonical code-tree, so that $S[a]$ $(1 \leq a \leq n)$ is the depth of the character a. Now, the depth and offset of any $a \in \Sigma$ is easily computed from the wavelet tree of S: the depth is just $S[a]$, while the offset is $rank_{S[a]}(S, a)$. Inversely, given a depth d and an offset o, the corresponding character is $select_d(S, o)$. The $\mathcal{O}(w^2)$-bit data structure of Gagie and Nekrich [9] converts in constant time pairs *(depth,offset)* into codes and vice versa (if $L = \mathcal{O}(w)$), whereas the multiary wavelet tree on S requires $n \log L + \mathcal{O}\left(\frac{n \log \log n}{\log_L n}\right)$ bits of space and completes encoding/decoding in time $\mathcal{O}\left(\frac{\log L}{\log \log n}\right)$. Under the restriction $1/\epsilon = \mathcal{O}(\mathrm{polylog}(n))$, the space is $\mathcal{O}(w^2) + n \log L + o(n)$ and the time is $\mathcal{O}(1)$. This is the key to the result of this section.

Theorem 1. *For any ϵ with $0 < \epsilon < 1/2$ and $1/\epsilon = \mathcal{O}(\mathrm{polylog}(n))$, and under the RAM model with computer word size w, so that the text to encode is of length $2^{\mathcal{O}(w)}$, we can store a prefix code with expected codeword length within an additive term ϵ of the minimum, using $\mathcal{O}\bigl(w^2 + n\log\log(1/\epsilon)\bigr)$ bits, such that encoding and decoding any character takes $\mathcal{O}(1)$ time.*

Proof. The data structure we have described achieves the given time bounds if we assume the text to encode is of length $m = 2^{\mathcal{O}(w)}$, as usual under the RAM model of computation, and thus $L = \mathcal{O}(w)$ enables constant-time encoding and decoding [9].

As for the space, we have shown it is $\mathcal{O}\bigl(w^2\bigr) + n\log L + o(n)$. To achieve the claim of the theorem we show that $H_0(S)$ is at most $\log(L - \lceil\log n\rceil + 1) + \mathcal{O}(1)$, so we can store S in $\mathcal{O}(n\log(L - \log n + 1) + n) = \mathcal{O}(n\log\log(1/\epsilon))$ bits.

To see this, consider S as two interleaved subsequences, S_1 and S_2, of length n_1 and n_2, with S_1 containing those lengths less than or equal to $\lceil\log n\rceil$ and S_2 containing those greater. Thus $nH_0(S) \le n_1H_0(S_1) + n_2H_0(S_2) + n$.

Since there are at most 2^ℓ codewords of length ℓ, assume we complete S_1 with spurious symbols so that it has exactly 2^ℓ occurrences of symbol ℓ. This completion cannot decrease $n_1H_0(S_1) = \sum_{1\le\ell\le\lceil\log n\rceil} \mathrm{occ}(\ell, S_1)\log\frac{n_1}{\mathrm{occ}(\ell,S_1)}$, as increasing some $\mathrm{occ}(\ell, S_1)$ to $\mathrm{occ}(\ell, S_1) + 1$ produces a difference of $f(n_1) - f(\mathrm{occ}(\ell, S_1)) \ge 0$, where $f(x) = (x + 1)\log(x + 1) - x\log x$ is increasing. Hence we can assume S_1 contains exactly 2^ℓ occurrences of symbol $1 \le \ell \le \lceil\log n\rceil$; straightforward calculation then shows that $n_1H_0(S_1) = \mathcal{O}(n_1)$.

On the other hand, S_2 contains at most $L - \lceil\log n\rceil$ distinct values, so $H_0(S_2) \le \log(L - \lceil\log n\rceil)$, unless $L = \lceil\log n\rceil$, in which case S_2 is empty and $n_2H_0(S_2) = 0$. Thus $n_2H_0(S_2) \le n_2\log(\lceil\log_\phi(1/\epsilon)\rceil + 1) = \mathcal{O}(n_2\log\log(1/\epsilon))$.

Combining both bounds, we get $H_0(S) = \mathcal{O}(1 + \log\log(1/\epsilon))$ and the theorem holds. $\square$

In other words, under mild assumptions, we can store a code using $\mathcal{O}(n\log\log(1/\epsilon))$ bits at the price of increasing the average codeword length by ϵ, and in addition have constant-time encoding and decoding. For constant ϵ, this means that the code uses just $\mathcal{O}(n)$ bits at the price of an arbitrarily small constant additive penalty over the shortest possible prefix code.

4   Multiplicative increase in expected codeword length

In this section we focus on a multiplicative rather than an additive penalty over the optimal prefix code, in order to achieve a sublinear-sized representation of the encoding, which still enables constant-time encoding and decoding.

Our main idea is to divide the alphabet into probable and improbable characters and to store information about only the probable ones. Given a constant $c > 1$, we use Milidiú and Laber's algorithm [15] to build a prefix code with maximum codeword length $L = \lceil\log n\rceil + \lceil 1/(c - 1)\rceil + 1$. We call a character's codeword *short* if it has length at most $L/c + 2$, and *long* otherwise. Notice there are at most $2^{L/c+3} - 1 = \mathcal{O}\bigl(n^{1/c}\bigr)$ characters with short codewords. Also,

although applying Milidiú and Laber's algorithm may cause some exceptions, characters with short codewords are usually more probable than characters with long ones. We will hereafter call *infrequent characters* those encoded with long codewords in the code of Milidiú and Laber.

We transform this length-restricted prefix code into a canonical code as described in Section 2; specifically, we sort the characters lexicographically within each depth. We use a dictionary data structure F due to Fredman, Komlós and Szemerédi [4] to store the indices of the characters with short codewords. This data structure takes $\mathcal{O}\big(n^{1/c} \log n\big)$ bits and supports membership queries in $\mathcal{O}(1)$ time, with successful queries returning the target character's codeword. We also build $\lfloor L/c \rfloor + 2$ arrays that together store the indices of all the characters with short codewords; for $1 \leq \ell \leq \lfloor L/c \rfloor + 2$, the ℓth array stores the indices of the characters with codewords of length ℓ, in lexicographic order by codeword. Again, we store the first codeword of each length in $\mathcal{O}\big(w^2\big)$ bits overall, following Gagie and Nekrich [9], such that it takes $\mathcal{O}(1)$ time to compute any codeword given its length and offset, and vice versa. With these data structures, we can encode and decode any character with a short codeword in $\mathcal{O}(1)$ time. To encode, we perform a membership query on the dictionary to check whether the character has a short codeword; if it does, we receive the codeword itself as satellite information returned by the query. To decode, we first find the codeword's length ℓ and offset j in $\mathcal{O}(1)$ time as described in Section 2. Since the codeword is short, $\ell \leq \lfloor L/c \rfloor + 2$ and the character's index is stored in the jth cell of the ℓth array. These data structures use a total of $\mathcal{O}\big(w^2 + n^{1/c} \log n\big)$ bits of space.

We replace each long codeword with *new* codewords: instead of a long codeword α of length ℓ, we insert $2^{L+1-\ell}$ new codewords $\alpha \cdot s$, where $\cdot$ denotes concatenation and s is an arbitrary binary string of length $L + 1 - \ell$. Figure 1 shows an example. Since $c > 1$, we have $n^{1/c} < n/2$ for sufficiently large n, so we can assume without loss of generality that there are fewer than $n/2$ short codewords; hence, the number of long codewords is at least $n/2$. Since every long codeword is replaced by at least two new codewords, the total number of new codewords is at least n. Since new codewords are obtained by extending all codewords of length $\ell > L/c + 1$ in a canonical code, all new codewords are binary representations of consecutive integers. Therefore the i-th new codeword equals to $\alpha_f + i - 1$, where α_f is the first new codeword. If a is an infrequent character, we encode it with the a-th new codeword, $\alpha_f + a - 1$. To encode a character a, we check whether a belongs to the dictionary F. If $a \in F$, then we output the codeword for a. Otherwise we encode a as $\alpha_f + a - 1$. To decode a codeword α, we read its prefix bitstring s_α of length $L + 1$ and compare s_α with α_f. If $s_\alpha \geq \alpha_f$, then $\alpha = s_\alpha$ is the codeword for $s_\alpha - \alpha_f + 1$. Otherwise, the codeword length of the next codeword α is at most $L/c + 1$ and α can be decoded as described in the previous paragraph. Notice we do not need to store the new codewords we just described, so the total space used is still $\mathcal{O}\big(w^2 + n^{1/c} \log n\big)$ bits.

Theorem 2. *For any constant $c > 1$, under the RAM model with computer word size w, so that the text to encode is of length $2^{\mathcal{O}(w)}$, we can store a prefix code*
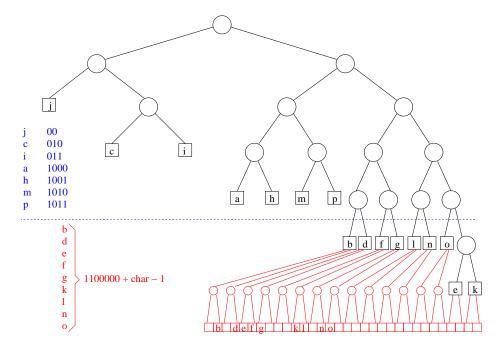
Fig. 1. An example with $n = 16$ and $c = 3$. The tree consisting of the nodes drawn as large circles and squares (in black) is the result of applying the algorithm of Milidiú and Laber on the original prefix code. Now, we set $L = 6$ according to our formula, and declare short the codeword lengths up to $\lfloor L/c \rfloor + 2 = 4$. Short codewords — i.e., those above the dashed line — are stored unaltered in a dictionary (in blue). Longer codewords — i.e., those below the dashed line — are changed: All are extended up to length $L + 1 = 7$ and reassigned a code according to their values in the contiguous slots of length 7 (in red).

with expected codeword length within c times the minimum in $\mathcal{O}\!\left(w^2 + n^{1/c} \log n\right)$ bits, such that encoding and decoding any character takes $\mathcal{O}(1)$ time.

Proof. The structure described throughout the section achieves the promised time and space bounds. We analyze now the expected codeword length.

By analysis of the algorithm by Milidiú and Laber [15] we can see that the codeword length of a character in their length-restricted code exceeds the codeword length of the same character in an optimal code by at most 1, and only when the codeword length in the optimal code is at least $L - \lceil \log n \rceil - 1 = \lceil 1/(c-1) \rceil$. Hence, the codeword length of a character encoded with a short codeword exceeds the codeword length of the same character in an optimal code by a factor of at most $\frac{\lceil 1/(c-1) \rceil + 1}{\lceil 1/(c-1) \rceil} \le c$. Every infrequent character is encoded with a codeword of length $L + 1$. Since the codeword length of an infrequent character in the length-restricted code is more than $L/c + 2$, its length in an optimal code is more than $L/c + 1$. Hence, the codeword length of a long character in our

code is at most $\frac{L+1}{L/c+1} < c$ times greater than the codeword length of the same character in an optimal code. Hence, the average codeword length for our code is less than c times the optimal one.   □

Again, under mild assumptions, this means that we can store a code with expected length within c times the optimum, in $\mathcal{O}\big(n^{1/c} \log n\big)$ bits and allowing constant-time encoding and decoding.

References

1. M. Adler and B. M. Maggs. Protocols for asymmetric communication channels. *Journal of Computer and System Sciences*, 63(4):573–596, 2001.
2. P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.
3. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007. Article 20.
4. M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $\mathcal{O}(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.
5. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.
6. T. Gagie. Compressing probability distributions. *Information Processing Letters*, 97(4):133–137, 2006.
7. T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.
8. T. Gagie. Dynamic asymmetric communication. *Information Processing Letters*, 108(6):352–355, 2008.
9. T. Gagie and Y. Nekrich. Worst-case optimal adaptive prefix coding. In *Proceedings of the Algorithms and Data Structures Symposium (WADS)*, pages 315–326, 2009.
10. E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38:933–967, 1959.
11. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
12. M. Karpinski and Y. Nekrich. A fast algorithm for adaptive prefix coding. *Algorithmica*, 55(1):29–41, 2009.
13. G. O. H. Katona and T. O. H. Nemetz. Huffman codes and self-information. *IEEE Transactions on Information Theory*, 22(3):337–340, 1976.
14. S. T. Klein. Skeleton trees for the efficient decoding of Huffman encoded texts. *Information Retrieval*, 3(4):315–328, 2000.
15. R. L. Milidiú and E. S. Laber. Bounding the inefficiency of length-restricted prefix codes. *Algorithmica*, 31(4):513–529, 2001.
16. A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.
17. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
18. N. Nakatsu. Bounds on the redundancy of binary alphabetical codes. *IEEE Transactions on Information Theory*, 37(4):1225–1229, 1991.

19. E. S. Schwarz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.
20. D. Sheinwald. On binary alphabetic codes. In *Proceedings of the Data Compression Conference (DCC)*, pages 112–121, 1992.