Simple Worst-Case Optimal Adaptive Prefix-Free Coding

Travis Gagie
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract

We give a new and simple worst-case optimal algorithm for adaptive prefix-free coding that matches Gagie and Nekrich's (2009) bounds except for lower-order terms, and uses no data structures more complicated than a lookup table.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Adaptive prefix-free coding, Shannon coding, Lookup tables

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.57

Funding *Travis Gagie*: Funded by NSERC Discovery Grant RGPIN-07185-2020.

1 Introduction

Suppose Alice has a deck of n cards, each marked with a character from a known alphabet of size σ, and she wants to send the sequence of the cards' characters to Bob over a noiseless binary channel. Moreover, suppose neither of them know in advance the frequencies of the distinct characters in the deck – perhaps it has just been shuffled – and Alice wants to encode each card's character before looking at the next card, such that Bob can recognize the last bit of that character's encoding when he receives it, and decode the character.

In 1973 Faller [1] proposed that Alice encode each card's character with a Huffman code [7] for the distribution of characters she has already seen, modified to assign codewords also to characters she has not yet seen. Thus, Alice encodes the ith character using a code that depends only on the first $i-1$ characters. Assuming Bob has already decoded the first $i-1$ characters when he receives the encoding of the ith, he can build the same code. Because the codewords in a Huffman code are prefix-free – no codeword is a prefix of another – he can then use that code to decode the ith character when he receives the last bit of its encoding. Since the codes adapt to the frequencies of the characters as more and more are seen, Faller's algorithm is said to perform adaptive Huffman coding or, more generally, *adaptive prefix-free coding*.

Building n Huffman codes from scratch takes $\Omega(n\sigma)$ time, but Faller showed how Alice and Bob can store a Huffman code such that after incrementing the frequency of a character, they can update the code in time proportional to the length of the codeword previously assigned to that character. It follows that Faller's algorithm runs in time proportional to the total length of the encoding of the sequence. Gallager [6] and Knuth [11] further developed Faller's algorithm in 1978 and 1985, respectively, and it is usually known as the FGK algorithm for their initials. It is commonly taught in courses on data compression and used in the classic UNIX utility `compact`, for example.

The same year Knuth published his work on the FGK algorithm, Vitter [15, 16, 17] gave a more sophisticated algorithm for adaptive Huffman coding. He showed it uses less than 1 more bit per character than Alice would use if she knew the characters' frequencies in advance, built a single Huffman code for them, sent it to Bob, and then used it to encode and send the sequence of characters in the deck. In other words, Vitter's algorithm uses at most about $n(H+1+\delta)$ bits and $O(n(H+1))$ time overall, where

$$H = \sum_{j=1}^{\sigma} \frac{n_j}{n} \lg \frac{n}{n_j} \leq \lg \sigma$$

is the entropy of the distribution $\frac{n_1}{n}, \ldots, \frac{n_\sigma}{n}$ of characters in the deck and $\delta \in [0, 1)$ is the redundancy of a Huffman code for that distribution. (When the distribution is dyadic – each frequency n_i is n divided by a power of 2 – then $\delta = 0$, and when nearly all the cards have the same character then δ is nearly 1.) Vitter's algorithm is also commonly taught in courses on data compression.

Vitter attributed to Chazelle an observation that the FGK algorithm uses at most about twice as many bits as using a single Huffman code for the characters' frequencies (the coefficient 2 can be reduced to about 1.44 using a result by Katona and Nemetz [10]), so the FGK algorithm also runs in $O(n(H + 1))$ time. In 1999 Milidiú, Laber and Pessoa [12] showed that with the FGK algorithm, Alice sends fewer than 2 more bits per character than she would with a single Huffman code for the characters' frequencies, or at most about $n(H + 2 + \delta)$ bits overall.

In 2003 Gagie [3, 4] showed that if we modify the FGK algorithm to perform adaptive Shannon coding instead of adaptive Huffman coding, then Alice sends at most about $H + 1$ bits per character. A Shannon code [13] is a prefix-free code that assigns any character with probability p a codeword of length at most $\left\lceil \lg \frac{1}{p} \right\rceil$ so, if Alice pretends she has seen each character once before she starts encoding the sequence $S[1..n]$ of the cards' characters, then she sends at most

$$\begin{aligned}
&\sum_{i=1}^{n} \left\lceil \lg \frac{i + \sigma - 1}{\mathrm{occ}(S[i], S[1..i-1]) + 1} \right\rceil \\
\leq\ & \sum_{i=1}^{n} \lg(i + \sigma - 1) - \sum_{i=1}^{n} \lg \big(\mathrm{occ}(S[i], S[1..i-1]) + 1\big) + n \\
<\ & \lg n! + \sigma \lg(n + \sigma) - \sum_{j=1}^{\sigma} \lg n_j! + n \\
=\ & \lg \binom{n}{n_1, \ldots, n_\sigma} + \sigma \lg(n + \sigma) + n
\end{aligned}$$

bits, where $\mathrm{occ}(S[i], S[1..i-1])$ is the frequency of $S[i]$ in the prefix $S[1..i-1]$ and $\binom{n}{n_1,\ldots,n_\sigma}$ is the number of distinct ways of arranging the cards in the deck. By Stirling's Approximation, $\lg \binom{n}{n_1,\ldots,n_\sigma} \leq nH + O(\sigma \log n)$, so with Gagie's algorithm Alice sends $n(H + 1) + o(n)$ bits as long as $\sigma \lg(n + \sigma) \in o(n)$.

In 2006 Karpinski and Nekrich [8, 9] gave a more sophisticated algorithm for adaptive Shannon coding, with which Alice sends $n(H + 1) + O(\sigma \log^2 n)$ bits and encodes S in $O(n)$ time, and Bob decodes it in $O(n(\log H + 1))$ time. Their algorithm uses canonical codes [14] and assumes Alice and Bob are working on word RAMs with $\Omega(\log n)$-bit words, as we do henceforth. Notice speeding up encoding on a word RAM is generally easier than speeding up decoding: for example, given a single prefix-free code whose longest codeword fits in a constant number of machine words, Alice can build an $O(\sigma)$-space lookup table that tells her the codeword for any character in constant time.

In 2009 Gagie and Nekrich [5] improved Karpinski and Nekrich's algorithm so that Bob decodes S also in $O(n)$ time, at the cost of increasing the bound on the total encoding length to $n(H + 1) + O(\sigma \log^{5/2} n)$. They also proved Alice must send $n\left(\lg \sigma + 1 - o(1)\right) \geq$

Table 1 The per-character bounds for the algorithms discussed, assuming $\sigma \in o\left(\frac{n^{1/2}}{\log n}\right)$, ignoring lower-order terms and omitting asymptotic notation.

authors	encoding length	encoding time	decoding time
FGK [1, 6, 11]	$H+2+\delta$	$H+1$	$H+1$
Vitter [15, 16, 17]	$H+1+\delta$	$H+1$	$H+1$
Gagie [3, 4]	$H+1$	$H+1$	$H+1$
KN [8, 9]	$H+1$	1	$\lg H+1$
GN [5]	$H+1$	1	1
new	$H+1$	1	1

$n(H+1-o(1))$ bits in the worst case, even when even when $\sigma \in \omega(1)$. To see why, suppose $\sigma = 2^{\lceil \lg f(n) \rceil} + 1$ for some function $f(n) \in \omega(1)$, so $\sigma \in \omega(1)$ and any prefix-free code for the alphabet assigns some character a codeword of length $\lceil \lg f(n) \rceil + 1 = \lg \sigma + 1 - o(1)$. For each i, the adversary chooses $S[i]$ to be a character with codeword length at least $\lg \sigma + 1 - o(1)$ in the code Alice will use to encode $S[i]$.

Gagie and Nekrich's algorithm is simultaneously worst-case optimal in terms encoding and decoding time and of encoding length, as long as $\sigma \in o\left(\frac{n}{\log^{5/2} n}\right)$, but it relies on constant-time predecessor queries on sets of $O(\log^{1/6} n)$ elements. These are theoretically possible on a word RAM with $\Omega(\log n)$-bit words [2] but very complicated and totally impractical. In this paper we give a new algorithm for adaptive prefix-free coding that is simple – it uses no data structures more complicated than a lookup table – but still uses $O(n)$ time for both encoding and decoding and $n(H+1) + O\left(\frac{n}{\log n} + \sigma^2 \log^2 n\right)$ bits, which is within lower-order terms of optimal when $\sigma \in o\left(\frac{n^{1/2}}{\log n}\right)$. Table 1 shows the per-character bounds of all the algorithms we have discussed here, assuming $\sigma \in o\left(\frac{n^{1/2}}{\log n}\right)$, ignoring lower-order terms and omitting asymptotic notation. We leave as future work finding a simple algorithm that is worst-case optimal even when σ is closer to n.

2 Algorithm

Before we describe our new algorithm, we briefly review how length-restricting a prefix-free code can speed up decoding. Our starting point is Gagie's [3, 4] observation that if we smooth the probability distribution $\frac{n_1}{n}, \ldots, \frac{n_\sigma}{n}$ by averaging it with the uniform distribution and apply Shannon's construction [13] to the result, then we obtain a prefix-free code with average codeword length

$$\sum_{j=1}^{\sigma} \frac{n_j}{n} \left\lceil \lg \frac{1}{\frac{1}{2}\left(\frac{n_j}{n} + \frac{1}{\sigma}\right)} \right\rceil < \sum_{j=1}^{\sigma} \frac{n_j}{n} \left(\lg \frac{n}{n_j} + 2 \right) = H + 2$$

when encoding S, and maximum codeword length at most $\lceil \lg \sigma \rceil + 1$.

In $O(\sigma)$ time we can build an $O(\sigma)$-space lookup table that, for any binary string of length $\lceil \lg \sigma \rceil + 1$, tells us which character's codeword is a prefix of that string and the length of that codeword. If we encode S by replacing each character by its codeword – which we can do in $O(n)$ time using an $O(\sigma)$-space lookup table that tells us the codeword for any character – then later we can decode S in $O(n)$ time by repeatedly taking prefixes of the encoding consisting $\lceil \lg \sigma \rceil + 1$ bits, looking up which character's codeword is a prefix of the encoding and the length of that codeword, and deleting the codeword from the beginning of the encoding. Unfortunately, we may use about $H+2$ bits per character.

If we take a weighted average of $\frac{n_1}{n}, \ldots, \frac{n_\sigma}{n}$ and the uniform distribution, however, then we can reduce the number of bits we use per character, at the cost of increasing the maximum codeword length and the space needed by the table. For example, if we assign weight $\frac{\lg n - 1}{\lg n}$ to $\frac{n_1}{n}, \ldots, \frac{n_\sigma}{n}$ and weight $\frac{1}{\lg n}$ to the uniform distribution before averaging them and applying Shannon's construction, then the average codeword length when encoding S is

$$\sum_{j=1}^{\sigma} \frac{n_j}{n} \left\lceil \lg \frac{1}{\frac{\lg n - 1}{\lg n} \cdot \frac{n_j}{n} + \frac{1}{\lg n} \cdot \frac{1}{\sigma}} \right\rceil < \sum_{j=1}^{\sigma} \frac{n_j}{n} \left(\lg \frac{n}{n_j} + \lg \frac{\lg n}{\lg n - 1} + 1 \right) < H + 1 + \frac{\lg e}{\lg n - 1}$$

and the maximum codeword length is at most $\lceil \lg(\sigma \lg n) \rceil = \lceil \lg \sigma + \lg \lg n \rceil$, so the lookup table takes $O(2^{\lg \sigma + \lg \lg n}) = O(\sigma \log n)$ space. Building this table takes $O(\sigma \log n)$ time.

We are now ready to describe our new algorithm. First, Alice encodes $S[1..\lceil \sigma \lg n \rceil]$ using a Shannon code C_0 for the uniform distribution, that assigns every character a codeword of length $\lceil \lg \sigma \rceil$. This takes her $O(\sigma \log n)$ time. Then, for $k \geq 1$, after encoding $S[1..k\lceil \sigma \lg n \rceil]$, Alice builds a Shannon code C_k for a weighted average of the distribution of characters she has encoded so far and the uniform distribution:

$$\frac{\lg n - 1}{\ln n} \cdot \frac{\operatorname{occ}\left(a_1, S[1..k\lceil \sigma \lg n \rceil]\right)}{k\lceil \sigma \lg n \rceil)} + \frac{1}{\lg n} \cdot \frac{1}{\sigma}, \ldots, \frac{\lg n - 1}{\ln n} \cdot \frac{\operatorname{occ}\left(a_\sigma, S[1..k\lceil \sigma \lg n \rceil]\right)}{k\lceil \sigma \lg n \rceil} + \frac{1}{\lg n} \cdot \frac{1}{\sigma},$$

where $\operatorname{occ}\left(a_j, S[1..k\lceil \sigma \lg n \rceil]\right)$ is the frequency in $S[1..k\lceil \sigma \lg n \rceil]$ of the jth character a_j in the alphabet. She builds an $O(\sigma)$-space lookup table for C_k that lets her encode $S[k\lceil \sigma \lg n \rceil + 1..(k+1)\lceil \sigma \lg n \rceil]$ in $O(\sigma \log n)$ time.

Because the codewords in C_0 have length $\lceil \lg \sigma \rceil$, Bob can build an $O(\sigma)$-space lookup table that lets him decode $S[1..\lceil \sigma \lg n \rceil]$ in $O(\sigma \log n)$ time. Then, for $k \geq 1$, after encoding $S[1..k\lceil \sigma \lg n \rceil]$, Bob builds the same Shannon code C_k that Alice used to encode $S[k\lceil \sigma \lg n \rceil + 1..(k+1)\lceil \sigma \lg n \rceil]$. Because the longest codeword in C_k has length at most $\lceil \lg \sigma + \lg \lg n \rceil$, Bob can build an $O(\sigma \log n)$-space lookup table that lets him decode $S[k\lceil \sigma \lg n \rceil + 1..(k+1)\lceil \sigma \lg n \rceil]$ in $O(\sigma \log n)$ time.

3 Analysis

For each $k \geq 0$, building C_k and the lookup tables for encoding and decoding with it takes Alice and Bob $O(\sigma \log n)$ time, and that cost is amortized over the $\lceil \sigma \lg n \rceil$ characters in $S[k\lceil \sigma \lg n \rceil + 1..(k+1)\lceil \sigma \lg n \rceil]$. Since encoding and decoding $S[k\lceil \sigma \lg n \rceil + 1..(k+1)\lceil \sigma \lg n \rceil]$ also takes $O(\sigma \log n)$ time, Alice and Bob each spend $O(n)$ time in total, or constant time per character in S.

Probably the most complicated aspect of our algorithm is the analysis showing the total length of the encoding is at most $n(H+1) + O\left(\frac{n}{\log n} + \sigma^2 \log^2 n\right)$. Consider that each character in S is encoded with $O(\sigma \log n)$ bits so, in particular, the first $\lceil \sigma \lg n \rceil$ occurrences of each distinct character are encoded with a total of $O(\sigma^2 \log^2 n)$ bits. Let I_j be the set of positions i such that character $S[i]$ of S is an occurrence of the jth character a_j in the alphabet but not one of a_j's first $\lceil \sigma \lg n \rceil$ occurrences. For $i \in I_j$, Alice encodes $S[i]$ using at most

$$\left\lceil \lg \left(\frac{1}{\frac{\lg n - 1}{\lg n} \cdot \frac{i-1}{\operatorname{occ}(a_j, S[1..i]) - \lceil \sigma \lg n \rceil}} \right) \right\rceil < \lg \frac{i - 1}{\operatorname{occ}(a_j, S[1..i]) - \lceil \sigma \lg n \rceil)} + 1 + \frac{\lg e}{\lg n - 1}$$

bits. Therefore, the total number of bits in the encoding is

$$\sum_j \sum \left\{ \lg \frac{i-1}{\operatorname{occ}(a_j, S[1..i]) - \lceil \sigma \lg n \rceil} \;:\; i \in I_j \right\} + n + O\left(\frac{n}{\log n} + \sigma^2 \log^2 n \right).$$

Notice $\sum_j \sum \{\lg(i-1) \ : \ i \in I_j\} \leq \lg n!$ and for each j,

$$\sum \left\{ \lg \left(\mathrm{occ}(a_j, S[1..i]) - \lceil \sigma \lg n \rceil \right) \ : \ i \in I_j \right\} = \lg(n_j - \lceil \sigma \lg n \rceil)! = \lg n_j! - O(\sigma \log^2 n) \, .$$

Therefore, the total number of bits in the encoding is at most

$$\begin{aligned} & \lg n! - \sum_{j=1}^{\sigma} \lg n_j! + n + O\left(\frac{n}{\log n} + \sigma^2 \log^2 n \right) \\ = \ & \lg \binom{n}{n_1, \ldots, n_\sigma} + n + O\left(\frac{n}{\log n} + \sigma^2 \log^2 n \right) \\ \leq \ & n(H+1) + O\left(\frac{n}{\log n} + \sigma^2 \log^2 n \right) . \end{aligned}$$

References

1 Newton Faller. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems and Computers*, pages 593–597, 1973.

2 Michael L Fredman and Dan E Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

3 Travis Gagie. Dynamic length-restricted coding. Master's thesis, University of Toronto, 2003.

4 Travis Gagie. Dynamic Shannon coding. In *Proceedings of the 12th European Symposium on Algorithms (ESA)*, pages 359–370, 2004.

5 Travis Gagie and Yakov Nekrich. Worst-case optimal adaptive prefix coding. In *Proceedings of the 11th Symposium on Algorithms and Data Structures (WADS)*, pages 315–326, 2009.

6 Robert Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

7 David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

8 Marek Karpinski and Yakov Nekrich. A fast algorithm for adaptive prefix coding. In *Proceedings of the International Symposium on Information Theory (ISIT)*, pages 592–596, 2006.

9 Marek Karpinski and Yakov Nekrich. A fast algorithm for adaptive prefix coding. *Algorithmica*, 55(1):29–41, 2009.

10 Gyula O H Katona and Tibor O H Nemetz. Huffman codes and self-information. *IEEE Transactions on Information Theory*, 22(3):337–340, 1976.

11 Donald E Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163–180, 1985.

12 Ruy Luiz Milidiú, Eduardo Sany Laber, and Artur Alves Pessoa. Bounding the compression loss of the FGK algorithm. *Journal of Algorithms*, 32(2):195–211, 1999.

13 Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

14 Jan van Leeuwen. On the construction of Huffman trees. In *Proceedings of the 3rd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 382–410, 1976.

15 Jeffrey Scott Vitter. Design and analysis of dynamic Huffman coding. In *Proceedings of the 26th Symposium on Foundations of Computer Science (FOCS)*, pages 293–302, 1985.

16 Jeffrey Scott Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34(4):825–845, 1987.

17 Jeffrey Scott Vitter. Algorithm 673: dynamic Huffman coding. *ACM Transactions on Mathematical Software*, 15(2):158–167, 1989.