A Fast Algorithm for Adaptive Prefix Coding[1]

Marek Karpinski* and Yakov Nekrich**

Abstract. In this paper we present a new algorithm for adaptive prefix coding. Our algorithm encodes a text S of m symbols in $O(m)$ time, i.e., in $O(1)$ amortized time per symbol. The length of the encoded string is bounded above by $(H+1)m + O(n \log^2 m)$ bits where n is the alphabet size and H is the entropy.
This is the first algorithm that adaptively encodes a text in $O(m)$ time and achieves an almost optimal bound on the encoding length in the worst case. Besides that, our algorithm does not depend on an explicit code tree traversal.

Keywords: Data Compression, Prefix-free Codes, Adaptive Coding.

1 Introduction

The prefix coding problem is a well-known and extensively studied algorithmic problem. Prefix coding is also one of the most popular data compression techniques nowadays. Different methods of prefix coding are used in a large number of well-known compression programs and formats, such as gzip, JPEG, and JPEG-LS. In this paper we present an algorithm for adaptive prefix coding that encodes in $O(m)$ time, where m is the number of symbols in the input string, and guarantees a good upper bound on the encoding length. This is the first algorithm with an upper bound on the encoding length whose running time depends only on the number of symbols in the input string but does not depend on the length of the encoded string (or on the alphabet size).

In the prefix coding problem, we are given a string $S = s_1 s_2 \ldots s_m$ over an alphabet $A = \{a_1, a_2, \ldots, a_n\}$. Each alphabet symbol a_i is assigned a codeword $c_i \in \{0,1\}^*$, so that no codeword is a prefix of another one. In the static prefix coding problem, the whole string S is known before the encoding starts, and all occurrences of the same symbol are encoded

[1] A preliminary version of this paper appeared in the Proceedings of the 2006 IEEE International Symposium on Information Theory (ISIT 2006).

* Dept. of Computer Science, University of Bonn. Work partially supported by a DFG grant, Max-Planck Research Prize, and IST grant 14036 (RAND-APX). Email: marek@cs.uni-bonn.de.

** Dept. of Computer Science, University of Bonn. Work partially supported by IST grant 14036 (RAND-APX). Email: yasha@cs.uni-bonn.de.

with the same codeword. For the static prefix coding, two passes over the data are necessary. The specification of the code is stored as a prefix of the encoded data of length $O(n \log n)$. In this paper, $\log n$ denotes the binary logarithm. In the case of adaptive prefix coding, each symbol s_i is encoded before the next symbol s_{i+1} is received. Assignment of codewords to symbols is based on their frequencies in the already encoded string $s_1 s_2 \ldots s_{i-1}$. Thus only one pass over the data is sufficient, and the codeword assignment is not stored with the encoded data. In adaptive coding, different occurrences of the same symbol may be encoded with different codewords.

The optimal algorithm for static prefix coding was described by Huffman [9]. The problem of optimal static prefix coding was investigated in a large number of papers; fast and efficient algorithms for encoding and decoding of static prefix codes are widely known (see e.g., [18], [2], [13], [11]). The classical algorithm of [9] is based on the construction of a code tree. Each leaf of the code tree corresponds to a symbol of the input alphabet, and each edge is labeled with 0 or 1. The codeword for a symbol $a_i \in A$ is a sequence of edge labels on the path from the root of the code tree to the leaf corresponding to a_i. To generate a codeword for some symbol $a_i \in A$, we must traverse the path from the root of the code tree to the leaf associated with a_i. Hence, the encoding time is proportional to the number of bits in the encoded string. However, we can easily reduce the encoding time to $O(m)$ by constructing a table of codewords. The bit length M of the string S encoded with the static Huffman code satisfies $Hm \leq M \leq (H + d)m$, where $H = \sum_{a \in A} \frac{\mathtt{occ}(a)}{m} \log(\frac{m}{\mathtt{occ}(a)})$ is the empirical zeroth-order entropy, $\mathtt{occ}(a)$ is the number of occurrences of symbol a in S, and d is the redundancy of the Huffman code. Redundancy d can be estimated using the symbol probabilities (e.g., $d < p_{max} + 0.086$, where p_{max} is the probability of the most frequent symbol, or $d < 1$; other estimates are also known, s. [3], [7]).

The first algorithm for adaptive Huffman coding was independently proposed by Faller [4] and Gallager [7], and later improved by Knuth [12]. In the work of Milidiu, Laber, and Pessoa [14] it was shown that the algorithm of Faller, Gallager, and Knuth, also known as the FGK algorithm, uses at most $2m$ more bits to encode S than the static Huffman algorithm. Vitter [20] presented an improved version of the adaptive Huffman coding that uses at most m more bits than the static Huffman coding. This means that FGK and Vitter's algorithm use respectively no more than $(H + 2 + d)m + O(n \log n)$ and $(H + 1 + d)m + O(n \log n)$ bits in the worst case, where d is the redundancy of the static code. Both the

FGK algorithm and the algorithm of Vitter require $\Theta(M)$ time to encode and decode S, where M is the number of bits in the encoding of S. Both algorithms maintain an optimal Huffman tree that is used for encoding and decoding. Recently, Gagie [6] described an adaptive coding method that is based on Shannon coding. The upper bound on the encoding length achieved by the method of [6] is $(H+1)m + O(n \log m)$ bits, for a parameter $l > \log n$. The algorithm of [6] also requires $\Theta(M)$ time for encoding and depends on maintaining a minimax tree [8]. It is interesting that although the method of [6] is based on sub-optimal Shannon codes it often achieves a better upper bound on the encoding length than adaptive coding methods of [20] and [12] that are based on maintaining the optimal Huffman code. All previous algorithms for adaptive prefix coding that guaranteed an upper bound on the length of encoding are based on maintaining the code tree. Therefore the best time that can be achieved by such algorithms is $O(M)$, for we must spend constant time for each output bit. Turpin and Moffat [19] describe an algorithm GEO for adaptive prefix coding. The main idea of the algorithm GEO is approximation of the frequency distribution of characters: if f_a is the frequency of a symbol a (i.e., a occurs f_a times), then approximate frequency of a is $f'_a = T^{\lfloor \log_T \mathtt{occ}(a_i) \rfloor}$ for $T = 2^{1/k}$ for an integer parameter k. That is, the symbol frequency f_a is replaced by the highest power of T that does not exceed T. Turpin and Mofat [19] show good upper bounds on the encoding length of GEO. Although the authors demonstrate the practical efficiency of GEO by practical experiments, the worst case asymptotic complexity of their encoding algorithm is $\Theta(M)$. The method presented in our paper is also based on approximation, but in a different way: we approximate probabilities instead of frequencies,and a different approximation formula is used.

In this paper we show that adaptive coding with a good upper bound on the encoding length can be achieved even if the input string is encoded in $O(m)$ time, i.e. in constant amortized time per symbol. We present an algorithm for adaptive coding that encodes in optimal $O(m)$ time. This is the first algorithm with running time independent of the alphabet size. The length of the encoding is bounded above by $(H+1)m + O(n \log^2 m)$. Thus our method achieves the upper bound on the encoding length, which is better than that of [20] and [12] if $n \ll m$. At the same time our algorithm is faster than previous algorithms for adaptive prefix coding, since all previous algorithms with a worst-case upper bound on the encoding length require $O(M) = O(mH)$ time to encode the string of m symbols with (zeroth-order) entropy H. We are able to achieve constant amor-

tized time only for the encoding algorithm. The decoding algorithm can work slightly faster than in $\Theta(M)$ time, because we use canonical codes. Details will be given in section 7.

Our approach is based on maintaining the Shannon code for the quantized symbol probabilities instead of using the exact symbol probabilities. This allows us to save time because the quantized weights are incremented less frequently. We also use the ideas of canonical prefix coding ([18], [2]) to avoid the explicit tree construction.

The rest of this paper has the following structure. We give a high-level description of the adaptive coding in section 2. In section 3 we describe the canonical Shannon codes. In section 4 quantized Shannon codes are presented. We show that dynamic quantized Shannon coding takes $O(m)$ time in section 5 and prove the upper bound on the encoding length in section 6. In section 6.1 we describe a modification of our algorithm for the case when the length of the text is unknown. Decoding methods are described in section 7. The computational model used in this paper is unit-cost RAM with word size $w = \Omega(m)$.

2 Preliminaries and Notation

We denote by $\mathtt{occ}(a, i)$ the number of occurrences of symbol a in $s_1 s_2 \ldots s_i$; $\mathtt{occ}(a) = \mathtt{occ}(a, m)$ denotes the number of occurrences of a in S. Further in this paper, n denotes the number of symbols in the alphabet plus one ($n = |A| + 1$).

We use the following high-level description of the adaptive coding process. Suppose that the prefix $s_1 s_2 \ldots s_{i-1}$ was already encoded and we are going to encode s_i. The following operations are performed:

1. Compute the prefix code for (NYT)$s_1 s_2 \ldots s_{i-1}$ by modifying the prefix code for (NYT)$s_1 s_2 \ldots s_{i-2}$ (using e.g., the algorithm of Vitter [20] or the algorithm described in this paper). Here (NYT) denotes a special symbol that is different from all symbols in the text.
2. If symbol s_i occurs in $s_1 s_2 \ldots s_{i-1}$, then the codeword for s_i is output. That is, we use the code for $s_1 s_2 \ldots s_{i-1}$ to encode s_i.
3. If s_i does not occur in $s_1 s_2 \ldots s_{i-1}$, then we output the codeword for (NYT) followed by a binary representation of the symbol s_i. The symbol (NYT) indicates the fact that s_i occurs for the first time; the binary representation of s_i takes at most $\lceil \log(n-1) \rceil$ bits.

In our high-level description we omitted some implementation details; see e.g., [12] for a more detailed description. The steps 2 and 3 in the

above description are straightforward and can be easily implemented in $O(1)$ time; in the rest of this paper we describe an efficient algorithm for the step 1.

3 Canonical Shannon Coding

The Shannon code was first described in the classical work of Shannon [17]. Let $\mathtt{occ}(a_i)/m$ be the empirical probability of symbol a_i in a string of length m. We assume that symbols are sorted according to their probabilities, $\mathtt{occ}(a_i)/m \geq \mathtt{occ}(a_j)/m$ for $i < j$, and let $\mathtt{cum}_i = \sum_{j=1}^{i-1} \mathtt{occ}(a_j)/m$ be the cumulative probability of the first $i-1$ most probable symbols. The codeword for the i-th symbol in a Shannon code consists of the first $\lceil \log(m/\mathtt{occ}(a_i)) \rceil$ bits of $\mathtt{cum}_i$.

Gagie [6] modified this definition in his paper on adaptive Shannon codes; his construction is based on minimax trees ([8]).

In this paper we construct adaptive codes with codeword length close to the codeword lengths in the Shannon code, but our algorithm is based on *canonical* codes ([2], [18]). A canonical code has the *numerical sequence property:* codewords with the same length are binary representations of consecutive integers. An example of a canonical code is given on Fig. 1.

For ease of description, we sometimes do not distinguish between a codeword c of length i and the number whose i-bit binary representation equals to c. If all codeword lengths are known, the codeword can be found using the standard canonical coding procedure. Let n_i be the number of codewords of length i. Let $\mathtt{base}[i]$ be the first codeword of length i. Then the j-th codeword of length i can be computed with the formula $\mathtt{base}[i] + j - 1$. The array $\mathtt{base}[]$ can be computed recursively: $\mathtt{base}[0] = 0$, $\mathtt{base}[i] = (\mathtt{base}[i-1] + n_{i-1}) \times 2$. Thus, if the length l of the codeword for symbol a is known, and the index of the codeword for a among all codewords of length l is known, the codeword for symbol a can be computed in constant time. For example, in the code described on Fig. 1, $n_1 = n_2 = 0$, $n_3 = 2$, $n_4 = 6$, and $n_5 = 8$. Then, $\mathtt{base}[1] = \mathtt{base}[2] = 0$, $\mathtt{base}[3] = 0$, $\mathtt{base}[4] = 4$, and $\mathtt{base}[5] = 20$. The codeword for symbol a_6 is the 4-th codeword of length 4; hence, the codeword for a_6 can be computed as $\mathtt{base}[4] + 3 = 4 + 3 = 7$ or $(0111)_2$ in binary.

4 Quantized Shannon Coding

For an arbitrary *quantization parameter* $q > 1$, let $\mathtt{occ}_q(a, i) = \lfloor \mathtt{occ}(a, i)/q \rfloor$ and $P_q(i) = \lceil i/q \rceil$. The main idea of our algorithm (further called *quan-*

a_1	000	a_7	1000
a_2	001	a_8	1001
a_3	0100	a_9	10100
a_4	0101	a_{10}	10101
a_5	0110	...	...
a_6	0111	a_{16}	11011

Fig. 1. An example of a canonical prefix code

tized Shannon coding) is the computation of the codeword length based on quantized empirical probabilities: if $\mathtt{occ}(s_i, i-1) \geq q$, the codeword length for s_i is $l(s_i) = \lceil \log(\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)}) \rceil$; if $0 < \mathtt{occ}(s_i, i-1) < q$, $l(s_i) = \lceil \log(i) \rceil$.

The length $l(a_i)$ of the codeword for symbol a_i in a quantized Shannon code is greater than or equal to the codeword length for the same symbol in the traditional Shannon code (denoted by $l^S(a_i)$). Since $P_q(i)q \geq i$ and $\mathtt{occ}_q(s_i, i-1)q \leq \mathtt{occ}(s_i, i-1)$, $\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)} = \frac{qP_q(i)}{q\mathtt{occ}_q(s_i, i-1)} \geq \frac{i}{\mathtt{occ}(s_i, i-1)}$. Therefore $l(a_i) \geq l^S(a_i)$, and $\sum 2^{-l(a_i)} \leq \sum 2^{-l^S(a_i)} \leq 1$ since Shannon code is a prefix code. Thus the codeword lengths of the quantized code satisfy the Kraft-McMillan inequality. Hence, it is possible to construct a prefix code in which a symbol a_i is assigned the codeword with length $l^S(a_i)$. It can be easily proven by induction that the procedure for the construction of a canonical code described in section 3 constructs a prefix code if codeword lengths satisfy the Kraft-McMillan inequality.

In our coding scheme the current text length i is replaced by a quantized length $P_q(i)$, and the current symbol frequency $\mathtt{occ}(s_i, i-1)$ is replaced by the quantized frequency $\mathtt{occ}_q(s_i, i-1)$. Due to this fact, we can update the code less frequently: in a text of length m, $P_q(i)$ is incremented $\lfloor m/q \rfloor$ times, and the frequency of symbol a is incremented only $\lfloor \mathtt{occ}(a)/q \rfloor$ times.

However, when $P_q(i)$ is incremented, the lengths of up to n codewords in the quantized code may change. To avoid having to update a large number of codewords, we use a method similar to the method used in [6]. We allow $l(s_i)$ to be slightly higher than $\lceil \log(\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)}) \rceil$, and maintain the following invariant:

1. If $\mathtt{occ}(s_i, i-1) \geq q$, $\lceil \log(\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)}) \rceil \leq l(s_i) \leq \lceil \log(\frac{P_q(i)+n}{\mathtt{occ}_q(s_i, i-1)}) \rceil$.
2. If $0 < \mathtt{occ}(s_i, i-1) < q$, $\lceil \log(i) \rceil \leq l(s_i) \leq \lceil \log(i+n) \rceil$

We store all symbols that occurred at least once in a doubly-linked list R. When the length of the codeword for a symbol a is updated, we set it to $\lceil \log(\frac{P_q(i)+n}{\mathtt{occ}_q(a, i-1)}) \rceil$ or $\lceil \log(i+n) \rceil$ (if $s_1 s_2 \ldots s_{i-1}$ was already encoded). We

guarantee that the length of the codeword for a is updated at least once when an arbitrary length n substring $s_j s_{j+1} \ldots s_{j+n}$ of S is encoded (even if $s_j s_{j+1} \ldots s_{j+n}$ does not contain a): when l_{max} elements are encoded, where l_{max} is the maximum codeword length, we delete the first $2l_{max}$ elements from R, update their lengths, and append the removed symbols at the end of R. The detailed description of the algorithm for updating a quantized Shannon code is given in the next section.

5 Adaptive Canonical Coding

In this section we describe the algorithm for updating the adaptive Shannon code. For each symbol a_j in the alphabet we keep track of the number of occurrences of a_j in $s_1 s_2 \ldots s_{i-1}$ where i is the length of the already encoded string. For every symbol $a_j \in A$, we store its codeword length $\texttt{len}(a_j)$ and its index among all codewords of length $\texttt{len}(a_j)$ denoted by $\texttt{ind}(a_j)$. Variable n_l indicates the number of codewords of length l; array $\texttt{base}[l]$ contains the first codeword of length l. We store all symbols $a_j \in A$ such that $\texttt{occ}(a_j, i-1) > 0$ in a list R. All symbols with codewords of length l are also kept in a doubly-linked list $C[l]$, so that the last codeword of length l can be found in constant time. We denote by l_{max} the maximum codeword length in a code.

When the length of the codeword for some symbol a changes from l_1 to l_2, we perform the following operations. Let $\texttt{ind}(a) = i$. Let a' be the symbol whose codeword is the last codeword of length l_1. If $l_1 > 1$, then the index of the last codeword with length l_1 is changed from n_{l_1} to i. We remove a' from the end of $C[l_1]$, replace a with a' in $C[l_1]$, and decrement n_{l_1} by one. Thus the new codeword for a' is the i-th codeword of length l_1 (i.e., the new codeword for a' is the old codeword for a). If $n_{l_1} = 1$, we simply set $n_{l_1} = 0$ and remove a from $C[l_1]$ so that the list $C[l_1]$ becomes empty. The new codeword for a is the last codeword of length l_2: we set $\texttt{ind}(a) = n_{l_2} + 1$, increment n_{l_2}, and add a at the end of $C[l_2]$. All operations above require $O(1)$ time. When the values of n_i are updated, we can re-compute the array $\texttt{base}[]$ in $O(l_{max})$ time using the recursive formula given in section 3. Thus changing the length of a codeword requires $O(l_{max})$ time. However, we can also change the lengths of l_{max} codewords in a code in $O(l_{max})$ time: firstly, we change the values of n_l and update the lists $C[l]$ in $O(l_{max})$ time; then, we re-compute the array $\texttt{base}[]$ in $O(l_{max})$ time. As an example, consider the code on Fig. 1 and suppose that the length of the codeword for a_6 has changed to 5. Then, we decrement n_4 and replace a_6 with a_8 in $C[4]$; the index of a_8

is changed to 4 (now the codeword for a_8 is $(0111)_2$). We make a_6 the last codeword of length 5: we set $\texttt{ind}[a_6] = 9$, $\texttt{len}[a_6] = 5$, and $n_5 = 9$. The array $\texttt{base}$ is changed accordingly: $\texttt{base}[5] = 18 = (10010)_2$ Now, the codeword for a_6 is $\texttt{base}[5] + \texttt{ind}[a_6] - 1 = (11010)_2$

Suppose that the string $s_1 s_2 \ldots s_{i-1}$ was already processed, and we encode symbol s_i. After encoding s_i we modify the code as follows:

- If the symbol s_i occurs for the first time, the length of the codeword for s_i is set to $\lceil \log(i+n+1) \rceil$, and s_i is inserted at the end of R.
- If $\texttt{occ}(s_i, i-1) > 0$ and $\texttt{occ}(s_i, i) \equiv 0 \pmod q$, then we remove s_i from R, re-compute the length of the codeword for s_i as $l(s_i) = \lceil \log \frac{P_q(i+1)+n}{\texttt{occ}_q(s_i,i)} \rceil$ and insert it at the end of R. We update the values of $\texttt{ind}(s_i)$, $\texttt{len}(s_i)$, n_{k_1}, n_{k_2}, and lists $C[k_1]$ and $C[k_2]$, where k_1 and k_2 are the lengths of the old and the new codewords for s_i. Finally, we update the array $\texttt{base}[t]$, $t = 1, 2, \ldots, l_{max}$. Those operations take $O(l_{max})$ time.
- Besides that, if $i \equiv 0 \pmod{l_{max}}$, then the first $k = 2l_{max}$ elements $a_{r_1}, a_{r_2}, \ldots, a_{r_k}$ are removed from R. The codeword length for each a_{r_j} is set to $l(a_{r_j}) = \lceil \log \frac{P_q(i+1)+n}{\texttt{occ}_q(a_{r_j},i)} \rceil$ (or $l(a_{r_j}) = \lceil \log(i+n+1) \rceil$ if $\texttt{occ}(a_{r_j}, i) < q$). Then, variables n_t and lists $C[t]$ for $t = 1, 2, \ldots, l_{max}$, and variables $\texttt{ind}(a_{r_j})$ and $\texttt{len}(a_{r_j})$ for $j = 1, \ldots, k$ are changed accordingly in $O(l_{max})$ time. Then, the array $\texttt{base}[]$ is re-computed. All those operations also take $O(l_{max})$ time. Finally, we re-insert $a_{r_1}, a_{r_2}, \ldots, a_{r_k}$ at the end of list R.

If we choose q so that $q \geq l_{max}$, then the quantized Shannon code can be updated in $O(1)$ amortized time. The maximal codeword length in a Shannon code does not exceed $\log(m+n) = O(\log m)$. Thus if we set the quantization parameter q to e.g., $\lceil \log m \rceil$, then the quantized Shannon code can be updated in $O(1)$ amortized time.

`Implementation Remark` The algorithm described above involves the computation of binary logarithms. We can compute $\lceil \log(x) \rceil$ for some number x in $O(1)$ time by computing the index of the most significant bit of x, $\texttt{msb}(x)$. The index of the most significant bit can be computed in $O(1)$ time using e.g., the method described in [5], Lemma 3. It is also possible to compute $\texttt{msb}(x)$ using AC^0 operations only; see the description of operation `LeftmostOne(X)` in [1], section 2. Let $\texttt{mask}(k) = \sum_{i=0}^{k-1} 2^k$, i.e. $\texttt{mask}(k)$ is the number with k rightmost bits equal to 1 and all other bits equal to 0. To compute $\lceil \log(x) \rceil$, we find $k = \texttt{msb}(x)$. If $(x \texttt{ AND } \texttt{mask}(k-1)) = 0$, where `AND` denotes bitwise AND operation, then $\lceil \log(x) \rceil = k$. Otherwise $\lceil \log(x) \rceil = k+1$.

6 Analysis of Adaptive Shannon Coding

Let V be the set of indices i such that $\mathtt{occ}_q(s_i, i-1) > 0$, and V' be the set of i such that $\mathtt{occ}(s_i, i-1) > 0$. We denote by $\widetilde{H}$ the sum $\sum_{i \in V} \log(\frac{i}{\mathtt{occ}(s_i, i-1)})$.

We start by estimating the total encoding length for all s_i, $i \in V$. We prove an upper bound on $\widetilde{H}$, and then show that the total encoding length of all s_i, $i \in V$, is not much bigger than $\widetilde{H}$.

Lemma 1. $\widetilde{H} = \sum_{i \in V} \log(\frac{i}{\mathtt{occ}(s_i, i-1)}) \leq Hm + O(n \log m)$

Proof:

$$\sum_{i \in V} \log(\frac{i}{\mathtt{occ}(s_i, i-1)}) \leq \sum_{i \in V'} \log(\frac{i}{\mathtt{occ}(s_i, i-1)}) <$$

$$\sum_{i=1}^{m} \log i - \sum_{i \in V'} \log(\mathtt{occ}(s_i, i-1)) \stackrel{①}{=} \sum_{i=1}^{m} \log i - \sum_{a \in A} \sum_{i=1}^{\mathtt{occ}(a)-1} \log i =$$

$$\log(m!) - \sum_{a \in A} \log(\mathtt{occ}(a)!) + \sum_{a \in A} \log(\mathtt{occ}(a))$$

We obtain the equality ① by observing that for any $a \in A$, $\mathtt{occ}(s_i, i-1)$, such that $s_i = a$ and $i \in V'$, assumes all values between 1 and $\mathtt{occ}(a) - 1$. Applying Stirling's formula, $x \log x - x \ln 2 < \log(x!) < x \log x - x \ln 2 + O(\log x)$. Therefore,

$$\widetilde{H} \leq m \log m - m \ln 2 + O(\log m) - \sum_{a \in A} (\mathtt{occ}(a) \log(\mathtt{occ}(a)) - \mathtt{occ}(a) \ln 2) +$$

$$O(n \log m) =$$

$$(-m \ln 2 + \sum_{a \in A} \mathtt{occ}(a) \ln 2) + (\sum_{a \in A} \mathtt{occ}(a) \log m - \sum_{a \in A} \mathtt{occ}(a) \log(\mathtt{occ}(a)))$$

$$+ O(n \log m) = \sum_{a \in A} \mathtt{occ}(a) \log(\frac{m}{\mathtt{occ}(a)}) + O(n \log m)$$

In the equation above we used the fact that $m = \sum_{a \in A} \mathtt{occ}(a)$. □

The compression loss caused by choosing quantized empirical probabilities instead of the exact empirical probabilities for all s_i, $i \in V$, can be estimated as follows:

Lemma 2. $\sum_{i \in V} \log(\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)}) \leq \widetilde{H} + O(nq \log m)$

Proof: Let $P_q(i)q = i + r_i$. Then,

$$\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)} = \frac{P_q(i)q}{\mathtt{occ}(s_i, i-1)} \frac{\mathtt{occ}(s_i, i-1)}{q\mathtt{occ}_q(s_i, i-1)} =$$
$$\frac{\mathtt{occ}(s_i, i-1)}{q\mathtt{occ}_q(s_i, i-1)} \frac{i + r_i}{\mathtt{occ}(s_i, i-1)} = \frac{\mathtt{occ}(s_i, i-1)}{q\mathtt{occ}_q(s_i, i-1)} \frac{i}{\mathtt{occ}(s_i, i-1)} (1 + \frac{r_i}{i})$$

Therefore,

$$\log(\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)}) = \log(\frac{i}{\mathtt{occ}(s_i, i-1)}) + \log(\frac{\mathtt{occ}(s_i, i-1)}{\mathtt{occ}_q(s_i, i-1)q}) + \log(1 + \frac{r_i}{i})$$

The two last terms in this sum can be estimated as follows. Since $0 \leq r_i < q$, $\log(1 + \frac{r_i}{i}) < \log(1 + \frac{q}{i}) < \frac{q}{i \ln 2}$. Summing up by all $i \in V$, $\sum_{i \in V} \frac{q}{i \ln 2} \leq \frac{q}{\ln 2} \sum_{i=1}^{m} (1/i) = O(q \log m)$. Let $\mathtt{occ}(a, i-1) = q \cdot \mathtt{occ}_q(a, i-1) + r_i(a)$ for $a \in A$. Then, $\mathtt{occ}(s_i, i-1) = \mathtt{occ}_q(s_i, i-1)q + r_i(s_i)$, and $\log(\frac{\mathtt{occ}(s_i, i-1)}{\mathtt{occ}_q(s_i, i-1)q}) = \log(1 + \frac{r_i(s_i)}{\mathtt{occ}_q(s_i, i-1)q}) < \frac{r_i(s_i)}{\mathtt{occ}_q(s_i, i-1)q \ln 2} < \frac{1}{\mathtt{occ}_q(s_i, i-1) \ln 2}$. Summing up by all $i \in V$, we obtain

$$\sum_{i \in V} \frac{1}{\ln 2 \cdot \mathtt{occ}_q(s_i, i-1)} \leq \sum_{a \in A} ((q/\ln 2) \sum_{j=1}^{\mathtt{occ}_q(a)-1} (1/j)) = O(nq \log m)$$

Therefore,

$$\sum_{i \in V} \log(\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)}) = \sum_{i \in V} (\log(\frac{i}{\mathtt{occ}(s_i, i-1)}) + \log(\frac{\mathtt{occ}(s_i, i-1)}{\mathtt{occ}_q(s_i, i-1)q}))$$
$$+ \log(1 + \frac{r_i}{i}) = \widetilde{H} + O(nq \log m)$$

□

In our method, the codeword for symbol s_i can use up to $\log(\frac{P_q(i)+n}{\mathtt{occ}_q(s_i, i-1)})$ bits instead of $\log(\frac{P_q(i)}{\mathtt{occ}_q(s_i, i-1)})$ bits (we ignore rounding up for a moment). We can estimate the penalty for those additional bits using the following lemma.

Lemma 3. *For any function $g : V \to \mathbb{N}$ and any $q > 0$, $\sum_{i \in V} \log(\frac{\lceil i/q \rceil + n}{g(i)}) \leq \sum_{i \in V} \log(\frac{\lceil i/q \rceil}{g(i)}) + nq \log(i_{max} + nq)$, where i_{max} is the maximum element in V.*

Proof:

$$\sum_{i \in V} \log(\frac{\lceil i/q \rceil + n}{g(i)}) = \sum_{i \in V} (\log(\frac{\lceil i/q \rceil}{g(i)}) + \log(1 + \frac{n}{\lceil i/q \rceil}))$$

Besides that,

$$\sum_{i\in V}\log(1 + \frac{n}{\lceil i/q\rceil}) \leq \sum_{i\in V}\log(1 + \frac{nq}{i}) = \log(\frac{\prod_{i\in V}(i+nq)}{\prod_{i\in V} i})$$

Suppose that V consists of elements $i_1, i_2, \ldots, i_v = i_{max}$, so that $i_1 < i_2 < \ldots < i_v$. Then, $i_{k+nq} \geq i_k + nq$ and

$$\frac{\prod_{i\in V}(i+nq)}{\prod_{i\in V} i} = \frac{(\prod_{k=1}^{v-nq}(i_k+nq))(\prod_{k=v-nq+1}^{v}(i_k+nq))}{(\prod_{k=nq+1}^{v} i_k)(\prod_{k=1}^{nq} i_k)} \leq \frac{(\prod_{k=1}^{v-nq} i_{k+nq})(\prod_{k=v-nq+1}^{v}(i_{max}+nq))}{(\prod_{k=1}^{v-nq} i_{k+nq})\cdot 1} = (i_{max}+nq)^{nq}$$

Hence,

$$\sum_{i\in V}\log(1 + \frac{n}{\lceil i/q\rceil}) \leq nq\log(i_{max}+nq)$$

and the statement follows. □

The total length of encoding s_i for all $i \in V$ is $\sum_{i\in V}\log(\frac{P_q(i)+n}{\mathtt{occ}_q(s_i,i-1)})$. It follows from Lemma 3 that $\sum_{i\in V}\log(\frac{P_q(i)+n}{\mathtt{occ}_q(s_i,i-1)}) \leq \sum_{i\in V}\log(\frac{P_q(i)}{\mathtt{occ}_q(s_i,i-1)}) + nq\log(m+nq)$. If $nq < m$, the last expression is $\sum_{i\in V}\log(\frac{P_q(i)}{\mathtt{occ}_q(s_i,i-1)}) + O(nq\log m)$.

It remains to estimate the length of encoding s_i, $i \notin V$.

Lemma 4. *The total length of encoding all s_i, $i \notin V$, is $O(nq\log m)$.*

Proof: Each symbol s_i, $i \in (V' \setminus V)$, i.e., each symbol s_i, such that s_i already occurred in $s_1 s_2 \ldots s_{i-1}$, but its number of occurrences is less than q, is encoded with at most $\log(i+n) = O(\log m)$ bits. Each symbol s_i that occurs for the first time is encoded with $O(\log m) + \log n = O(\log m)$ bits. Since the total number of symbols s_i that occurred less than q times in $s_1 s_2 \ldots s_{i-1}$ is $O(nq)$, the statement of the Lemma follows. □

Theorem 1. *The quantized Shannon coding uses $(H+1)m + O(nq(\log(m+nq)))$ bits.*

Proof: All symbols s_i, such that $i \notin V$ can be encoded with $O(nq\log m)$ bits by Lemma 4. All symbols s_i, $i \in V$, require $\sum_{i\in V}\lceil\log(\frac{P_q(i)+n}{\mathtt{occ}_q(s_i,i-1)})\rceil \leq \sum_{i\in V}\log(\frac{P_q(i)+n}{\mathtt{occ}_q(s_i,i-1)}) + m$ bits. Using Lemmas 1, 2, 3, and the discussion above, $\sum_{i\in V}\log(\frac{P_q(i)+n}{\mathtt{occ}_q(s_i,i-1)}) = Hm + O(nq(\log(m+nq)))$. Therefore the

total encoding length does not exceed $m(H+1) + O(nq(\log(m+nq)))$ bits. □

The main result of this paper follows if we substitute $q = \Theta(\log m)$ in the above theorem.

Corollary 1. *Quantized Shannon coding with quantization parameter $q = \Theta(\log m)$ uses $(H+1)m + O(n \log^2 m)$ bits, and can be implemented in $O(m)$ time if $m \geq n$, where m is the text length and n is the extended alphabet size.*

6.1 Adaptive Coding of a String of Unknown Length

In the Corollary 1 we set $q = \Theta(\log m)$, i.e. we assumed that the length m of the input string is known in advance. If the input length is unknown, we can set $q = \lceil \log m_1 \rceil$ for some tentative length m_1; after m_1 input symbols are encoded, we set the tentative input length to some $m_2 \gg m_1$ and $q = \lceil \log m_2 \rceil$. Thus we produce a sequence of tentative input length until the input string is encoded completely. For instance, we can set $q_2 = \lceil \log m_1 \rceil$ for $m_1 = n^2$. When the first m_1 symbols are encoded, we set $q_2 = \lceil \log m_2 \rceil$ for $m_2 = 2n^2$. In this way we produce a sequence of tentative input lengths $m_1, m_2, \ldots, m_k$ with $m_1 = n^2$ and $m_i = 2m_{i-1}$ for $i > 1$. When the first m_i input symbols are encoded, we set $q_{i+1} = \lceil \log m_{i+1} \rceil$ and update the code by re-computing codeword lengths of all already encoded symbols s_j, $\mathtt{occ}(s_j, m_i) > 0$, as $l(s_j) = \lceil \log \frac{P_{q_{i+1}}(m_i+1)+n}{\mathtt{occ}_{q_{i+1}}(s_j, m_i)} \rceil$. When the code is updated, we must re-compute the lengths of at most n codewords in $O(n)$ time. Since $m_{i+1} - m_i \geq n^2$, re-computing the code takes $o(1)$ amortized time. It is easy to check that the maximum quantization parameter $q_{\max}$ used by this modified algorithm satisfies $q_{\max} = O(\log m)$. Therefore the modified algorithm can also be implemented in $O(m)$ time, and the upper bound on the encoded string length is $(H+1)m + O(n \log^2 m)$ bits.

7 Decoding of a Quantized Shannon Code

Since quantized Shannon code is a canonical code, decoding of a quantized Shannon code can work faster than tree-based coding decoding algorithms. Below we describe two simple decoding methods that work in $O(m \log\log m)$ and $O(m \log H)$ time respectively.

The decoding algorithms described below rely heavily on the algorithms for decoding of canonical prefix codes [15].

The decoding algorithm maintains the quantized Shannon code as described in section 5. Additionally, we maintain a two-dimensional array

$\texttt{sym}[l,i]$, $1 \le l \le l_{max}$, $1 \le i \le n$. For every pair (l,i), such that $1 \le l \le l_{max}$ and $1 \le i \le n_l$, $\texttt{sym}[l,i]$ contains the symbols that corresponds to the i-th codeword of length l. For $i > n_l$, $\texttt{sym}[i,l]$ is undefined. The array $\texttt{sym}$ uses $O(nl_{max}) = O(n \log m)$ words.

Let $\texttt{Sbase}[i] = \texttt{base}[i] \ll (w-i)$, where w is the length of the machine word. That is, $\texttt{Sbase}[i]$ is obtained by shifting $\texttt{base}[i]$ $(w-i)$ bits to the left. It can be easily checked that the values stored in array $\texttt{Sbase}$ grow monotonously, so that $\texttt{Sbase}[i] \le \texttt{Sbase}[i+1]$ for $1 \le i < l_{max}$. The decoding algorithm reads a sequence of bits from the input stream and transforms it into a sequence of symbols. Let B denote the buffer variable that contains the next w not yet decoded bits from the input stream. If the length ℓ of the next codeword is known, the next symbol can be decoded in constant time: The index of the next codeword is $i = (B \gg \ell) - \texttt{base}[\ell]$, and the next symbol is $s = \texttt{sym}[\ell, i]$. Hence, the only time-consuming operation is computing the length of the next codeword in the input stream.

We can obtain the length of the next codeword by comparing the value of B with elements of $\texttt{Sbase}$: if the length of the next codeword is ℓ, then $\texttt{Sbase}[\ell] \le B < \texttt{Sbase}[\ell+1]$. If the elements of $\texttt{Sbase}$ are stored in a balanced tree, such as the AVL tree, then ℓ can be found in $O(\log(l_{max})) = O(\log \log m)$ time. Thus we obtain a $O(m \log \log m)$ time decoding algorithm.

A more efficient algorithm can be obtained with the following simple technique. We consecutively compare B with $\texttt{Sbase}[l_1], \texttt{Sbase}[l_2], \ldots \texttt{Sbase}[l_i], \ldots$, where l_1 is the minimum codeword length and $l_i = l_{i-1} + 2^{i-1}$ for $i > 1$. Clearly, we can find i, such that $\texttt{Sbase}[l_{i-1}] < B \le \texttt{Sbase}[l_i]$, in $O(\log \ell)$ time. After that, the codeword length ℓ can be found by binary search in $O(\log(l_i - l_{i-1})) = O(\log \ell)$ time. As shown in Corollary 1, the total encoding length is $L = (H+1)m + O(n \log^2 m)$. Hence, the average codeword length is $l_{AV} = O(H)$ for $n < m/\log^2 m$. By Jensen's inequality, $\frac{1}{m}\sum_{i=1}^{m} \log(\texttt{len}(s_i)) \le \log(l_{AV}) = O(\log H)$. Therefore the total decoding time is $O(m \log H)$.

Acknowledgments

We thank Larry Larmore for interesting remarks and discussions and the anonymous reviewers for their comments and suggestions. Special thanks are to the anonymous reviewer for his suggestions on the implementation of the decoding methods.

References

1. A. Andersson, P. B. Miltersen, M. Thorup, "Fusion Trees can be Implemented with AC0 Instructions Only", Theor. Comput. Sci. 215(1999), 337-344.
2. J.B. Connell , "A Huffman-Shannon-Fano Code", Proc. of IEEE 61(1973), 1046-1047.
3. R. M. Capocelli, A. De Santis, "New Bounds on the Redundancy of Huffman Codes". IEEE Trans. Information Theory 37(1991), 1095-1104.
4. N. Faller, "An Adaptive System for Data Compression", Proc. 7th Asilomar Conference on Circuits, Systems, and Computers (1973), 593-597.
5. M. L. Fredman, D. E. Willard, "Surpassing the Information Theoretic Bound with Fusion Trees", J. Comput. Syst. Sci. 47(1993), 424-436.
6. T. Gagie, "Dynamic Shannon Coding", Proc. the 12th European Symposium on Algorithms (2004),LNCS 3221, 359-370; *see also* Information Processing Letters 102(2007), 113-117.
7. R. G. Gallager, "Variations on a Theme by Huffman", IEEE Trans. on Information Theory 24(1978), 668-674.
8. M. C. Golumbic, " Combinatorial Merging", IEEE Trans. Computers 25(1976), 1164-1167.
9. D.A.Huffman, "A Method for Construction of Minimum Redundancy Codes", Proc. IRE 40(1951), 1098-1101.
10. G.H. O. Katona , T. O. H. Nemetz, " Huffman Codes and Self-information", IEEE Trans. on Information Theory 22(1976), 337-340.
11. S. T. Klein, "Space- and Time-Efficient Decoding with Canonical Huffman Trees", Proc. the 8th Annual Symposium on Combinatorial Pattern Matching (1997), LNCS 1264, 65 - 75.
12. D. E. Knuth, "Dynamic Huffman Coding", J. Algorithms 6(1985), 163-180.
13. D.A. Lelewer, D.S. Hirschberg, "Data Compression", ACM Computing Surveys 19(1987), 261-296.
14. R. L. Milidiu, E. S. Laber, A. A. Pessoa, " Bounding the Compression Loss of the FGK Algorithm", J. Algorithms 32(1999), 195-211.
15. A. Moffat, A. Turpin, "On the Implementation of Minimum-Redundancy Prefix Codes", IEEE Transactions on Communications, 45(1997), 1200-1207.
16. L. Rueda, B. J. Oommen, " A Fast and Efficient Nearly-Optimal Adaptive Fano Coding Scheme" Information Sciences 176(2006), 1656-1683.
17. C.E. Shannon, "A Mathematical Theory of Communication", Bell System Technical Journal 27(1948), 379-423, 623-656.
18. E.S. Schwartz, B. Kallick, "Generating a Canonical Prefix Encoding", Comm. of the ACM 7(1964), 166-169.
19. A. Turpin, A. Moffat, "On-line adaptive canonical prefix coding with bounded compression loss", IEEE Trans. on Information Theory, 47(2001), 88–98.
20. J. S. Vitter, " Design and Analysis of Dynamic Huffman Codes", J. ACM 34(1987), 825-845.