Optimal Alphabet Partitioning for Semi-Adaptive Coding of Sources of Unknown Sparse Distributions

Dan Chen* Yi-Jen Chiang† Nasir Memon‡ Xiaolin Wu§

Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201

Abstract

Practical applications that employ entropy coding for large alphabets often partition the alphabet set into two or more layers and encode each symbol by using some suitable prefix coding for each layer. In this paper we formulate the problem of optimal alphabet partitioning for the design of a two layer semi-adaptive code and give a solution based on dynamic programming. However, the complexity of the dynamic programming approach can be quite prohibitive for a long sequence and a very large alphabet size. Hence, we give a simple greedy heuristic algorithm whose running time is linear in the number of symbols being encoded, irrespective of the underlying alphabet size. We give experimental results that demonstrate the fact that superior prefix coding schemes for large alphabets can be designed using our approach as opposed to the typically ad-hoc partitioning approach applied in the literature.

1 Introduction

For various reasons, some of them technical and some not, the most widely used entropy coding technique today is Huffman coding [2]. Although adaptive versions of Huffman's algorithm have been known for many years [5, 12], due to complexity issues, Huffman coding is most widely used in a static or semi-adaptive form. Static Huffman coding can be used when there is some knowledge about the statistics of the data being coded. However, when this is not the case, a semi-adaptive Huffman code

*dchen01@utopia.poly.edu. Research supported by NSF Grant ACI-0118915.

†yjc@poly.edu. Research supported in part by NSF Grant ACI-0118915, NSF CAREER Grant CCR-0093373, and NSF ITR Grant CCR-0081964.

‡memon@poly.edu. Research supported in part by NSF Grant ACI-0118915 and NSF Grant CCR-0208678.

§xwu@poly.edu. Research supported in part by NSF Grant CCR-0208678.

can be used which makes two passes through the data. In the first pass, statistics are collected to aid the construction of the Huffman code and the actual coding is only accomplished in the second pass. The specific Huffman code designed for the data is included in the data using some canonical representation technique. Most compression standards like JPEG, GIF, gzip, etc. include a semi-adaptive Huffman code as one of the supported, and indeed most widely used, options for entropy coding.

Although the simplicity and effectiveness of static and semi-adaptive Huffman coding provide a compelling rationale for their deployment in many applications, complexity issues start becoming serious concerns as the underlying alphabet size of the data being encoded start getting very large. This is due to the fact that for large alphabet sizes, a Huffman code for the entire alphabet requires an unduly large code table. This is the case even for a semi-adaptive technique where only the codewords for symbols that occur in the data being compressed need to be stored. An excessively large Huffman table can lead to multiple problems. First of all, in the semi-adaptive case, a larger code would require more bits to represent thereby reducing the efficiency of Huffman coding. Additionally, in both the static and semi-adaptive case, large Huffman tables can lead to serious difficulties in a hardware implementation of the codec.

1.1 Alphabet Partitioning

One way to deal with the problem of entropy coding large alphabets is to partition the alphabet into smaller sets and use a product code architecture where a symbol is identified by a set number and then by the element within the set. This approach has also been called *amplitude partitioning* [10] in the literature. One special case of the product code arises when the cardinality of each partition is 2^k for some k and exactly k bits are employed to identify any given element in that set. This strategy has been employed in many data compression standards in the context of Huffman coding and has also been called *Modified Huffman Coding*. For example, a modified Huffman coding procedure specified by the JPEG standard [8] is used for encoding DC coefficient differences. Here, each prediction error (or DC difference in the lossy codec) is classified into a "magnitude category" and the label of this category is Huffman coded. Since each category consists of a multiple number of symbols, uncoded "extra bits" are also transmitted which identify the exact symbol (prediction error in the lossless codec, and DC difference in the lossy codec) within the category.

Other well known standards like the CCITT Group 3 and Group 4 facsimile compression standards [3] and the MPEG video compression standards [7] also use a similar modified Huffman coding procedure to keep the Huffman table small relative to the underlying alphabet size they are entropy coding.

Note that the alphabet partitioning approach can also be applied recursively. That is, the elements within each set can be further partitioned in subsets and so on. One popular way of doing this is used when encoding floating-point numerical data that arise in compression of scientific applications. Here, for example, the 32-bit floating-point numbers to be encoded are viewed as a sequence of four bytes and separate entropy codes are designed for each of the four bytes. This, clearly can be viewed as recursive alphabet partitioning. The first level partitions the numbers into 256 different sets and the elements within each set are further partitioned into 256 sets and so on. Similar comments apply if the numbers are 64 bits or 80 bits or any other representation and if the partitioning is done based on bytes or nibbles or 16-bit chunks, etc.. Another more sophisticated, but nevertheless ad-hoc, recursive partitioning approach called *group partitioning* is presented in [14].

1.2 Our Contribution - Optimal Alphabet Partitioning

Although the modified Huffman coding procedures designed by JPEG, CCITT and other standards are quite effective, they are clearly ad-hoc in nature. For example, the design of the JPEG Huffman code is guided by the fact that the underlying symbols are DCT coefficients and are well modeled by a Laplacian or a generalized Gaussian distribution [11]. This knowledge is exploited in designing a suitable modified Huffman code. The exponentially decaying pmf's of quantized DCT coefficients are matched with partitions of exponentially increasing size, resulting in each partition being assigned a roughly equal probability mass.

Given the ad-hoc nature of alphabet partitioning techniques in the literature, our work is motivated by the following question: given a finite input sequence S drawn from a very large alphabet $\mathcal{Z}$ but with an unknown distribution, how does one design an optimal semi-adaptive modified Huffman code? Or more generally, how does one optimally partition the alphabet into sets, such that any symbol is encoded by an entropy code (say, a Huffman code) of the index of the partition to which it belongs followed by a suitable encoding of its rank within the partition. By optimal here, we mean a code that minimizes the number of bits needed to encode the symbol plus the number of overhead bits to describe the code.

The rest of this paper is organized as follows. In the next section we formulate the problem of optimal alphabet partitioning for the design of a two-layer entropy code and give a solution based on dynamic programming. However, the complexity of the dynamic programming approach can be quite prohibitive for a long sequence and a large alphabet size. Hence in Section 3 we give a greedy heuristic whose running time is $O(N)$ where N is the number of distinct symbols being encoded. Finally, in Section 4 we give experimental results that demonstrate the fact that superior semi-adaptive entropy coding schemes for large alphabets can be designed using our approach as opposed to the typically ad-hoc partitioning approach applied in the literature.

1.3 Related Work

The problem of large alphabet size also arises in adaptive single-pass entropy coding, for example, adaptive arithmetic coding. Here in addition to the large table size, the lack of sufficient samples in order to make reliable probability estimations becomes a more serious concern. Specifically, it is well known that the *zero frequency* problem can lead to serious coding inefficiency. The widely known PPM text compression algorithm [1] allocates explicit codes only to symbols that have occurred in the past and uses a special escape symbol to code a new symbol. In the work of Zhu *et. al.* [13], this approach is extended further by the use of a dual symbol set adaptive partitioning scheme, where the first set contains all the symbols that are likely to occur and the second set contains all the other symbols. Only the symbols in the first set are explicitly coded. They also give an adaptive procedure for moving symbols between the two sets.

More recently Yang and Jia [4] have given a universal scheme for coding sources with large and unbounded alphabets, which employs multi-level arithmetic coding. They dynamically construct a tree-based partition of the alphabet into small subsets and encode a symbol by its path to the leaf node that represents the subset of the symbol and by the index of the symbol within this subset. They prove that their code is universal, that is, it asymptotically achieves the source entropy.

The problem of minimizing space and time complexity when constructing prefix codes for large alphabets has been extensively studied in a series of papers by Moffat, Turpin *et. al.*. For example, see [6] and the references therein.

2 Optimal Alphabet Partitioning by Dynamic Programming

Our problem is to code a finite input sequence S drawn from a very large, possibly infinite, alphabet $\mathcal{Z}$ but with an unknown distribution. Since the size of alphabet is much larger than the input sequence length, $|\mathcal{Z}| >> |S|$, one-pass universal coding approach may not work well in practice due to the sample sparsity, i.e., the input sequence does not supply enough samples to reach a good estimate of the source distribution. Instead, we propose a two-pass and two-layer coding approach based on the minimum description principle [9].

We assume that the alphabet $\mathcal{Z}$ is ordered, which may be unbounded. Suppose that the input sequence consists of N distinct values: $z_1 < z_2 < \cdots < z_N$. Central to our two-layer code design is the concept of optimal alphabet partition. The range of alphabet $\mathcal{Z}$ is partitioned into M intervals: $(-\infty, z_{q_1}], (z_{q_1}, z_{q_2}], \cdots, (z_{q_{M-1}}, \infty)$. Note the M-cell partition of $\mathcal{Z}$ is defined by $M-1$ existing symbol values of S, indexed by $q_1 < q_2 < \cdots q_{M-1}$. Consequently, the symbols of input sequence S is partitioned into M subsets: $S(q_{i-1}, q_i] = \{z_j | j \in (q_{i-1}, q_i]\}$, $1 \le i \le M$, where $q_0 = 0$ and $q_M = N$ by convention. This partition constitutes the first layer of the code. Given an alphabet partition, the self entropy of the first layer is

$$H(q_1, q_2, \cdots, q_{M-1}) = -\sum_{i=1}^{M} P(S(q_{i-1}, q_i]) \log P(S(q_{i-1}, q_i]). \tag{1}$$

This is the average bit rate required to identify the subset membership of an input symbol in the alphabet partition.

We introduce a so-called operationally optimal Golomb code to resolve the remaining uncertainty within a cell of the alphabet partition, which is the second layer of our code. Given a data set $S(q_{i-1}, q_i]$, we find the optimal orientation of the Golomb code since the pmf $P(z_j | j \in (q_{i-1}, q_i])$ can have an increasing or decreasing tendency in j, and determine the optimal Golomb parameter that yields the minimum code length. Note that the Golomb coding scheme is only operationally and not information-theoretically optimal in the sense that we settle with the best the Golomb code can do with respect to the given pmf $P(z_j | j \in (q_{i-1}, q_i])$ which may deviate significantly from the exponential distribution. Our design decision here is purely practical because Golomb code is known to admit a fast implementation and has only one parameter, which means small side information in a two-pass coding scheme.

The proposed two-layer coding scheme is semi-adaptive. The encoder first scans the input sequence and collects the statistics to optimize the alphabet partition as described by the following algorithm. It then sends side information on alphabet partition, namely values $z_{q_1}, z_{q_2}, \cdots, z_{q_{M-1}}$, and the M optimal Golomb parameters and orientations, one for each alphabet interval.

Now we present a dynamic programming algorithm for optimal alphabet partition. Let $L(z_j | j \in (q_{i-1}, q_i])$ be the length of the operationally optimal Golomb code of the set $\{z_j | j \in (q_{i-1}, q_i]\}$. Let $R_k(0, n]$ be the overall code length produced by the two-layer coding scheme under the optimal k-cell alphabet partition of the subset $\{z | z < z_n\}$ of the input data set. Namely,

$$R_k(0, n] = \min_{0=q_0<q_1<\cdots<q_k=n} \left\{ \sum_{i=1}^{k} L(z_j | j \in (q_{i-1}, q_i]) + H(q_1, q_2, \cdots, q_{k-1}) |S(0, n]| + k\epsilon \right\}. \tag{2}$$

Here we assume that ϵ bits of side information per partition cell are used to specify the orientation and optimal Golomb parameter. Our objectives are to find

$$R_M(0, N], \quad M = \arg\min_k R_k(0, N], \tag{3}$$

and to construct the underlying alphabet partition given by $0 = q_0 < q_1 < \cdots < q_M = N$. The dynamic programming optimization procedure is based on the recursion:

$$R_k(0, n] = \min_{0<q<n} \{R_{k-1}(0, q] + L(z_j | j \in (q, n]) - |S(q, n]| \log P(S(q, n])\}, \tag{4}$$

After precomputing

$$R(a, b] = L(z_j | j \in (a, b]) - |S(a, b]| \log P(S(a, b]), \quad 0 < a < b \leq N \tag{5}$$

for all possible $O(N^2)$ subsets, the dynamic programming algorithm proceeds as described by the following pseudo code.

/* Initialization */
$q_0 \equiv 0$;
$q_M \equiv N$;
$m := 1$;
$R_1(0, N] = L(z_j | j \in (0, N])$;

/* Minimize the description length */
repeat
    $m := m + 1$;
    for $n := m$ to N do
        $R_m(0, n) := \min_{0<q<n}\{R_{m-1}(0, q] + L(z_j | j \in (q, n]) - |S(q, n]| \log P(S(q, n])\}$;
        $Q_m(n) := \arg\min_{0<q<n}\{R_{m-1}(0, q] + L(z_j | j \in (q, n]) - |S(q, n]| \log P(S(q, n])\}$;
until $R_m(0, N] + m\epsilon > R_{m-1}(0, N] + (m-1)\epsilon$.
/* Construct the alphabet partition */
$M := m - 1$;
for $m := M$ downto 2 do
    $q_{m-1} = Q_m(q_m)$;
output final M-cell optimal alphabet partition $(q_0, q_1, \cdots, q_M)$.

The complexity of the above dynamic programming algorithm is clearly $O(MN^2)$.

It should be noted that the algorithm we have presented is quite general and can be adapted for different types of coding strategies. For example, in the above algorithm we use Golomb codes for encoding the rank of a symbol within a partition. However, we could just use fixed length coding for this purpose as well (as done by the JPEG modified Huffman code) and the above algorithm has to be quite trivially modified to yield a suitable optimal partitioning of the alphabet.

3 Greedy Heuristic

The running time of the dynamic programming algorithm as given in the previous section is $O(MN^2)$. When coding a large number of symbols (drawn from a large alphabet), say tens of thousands, this is clearly prohibitive, even though the cost is only incurred by the encoder and not the decoder. In this section we present a sub-optimal greedy heuristic for partitioning the alphabet, which runs in time linear in the number of distinct symbols on the source sequence, that is, in $O(N)$ time. In the next section we give experimental results showing that the greedy heuristic yields results quite close to the optimal and hence can be an attractive alternative in many applications.

The greedy algorithm starts by first sort the distinct symbols that have occurred in the source sequence in increasing order. Then we make a linear scan of the symbols starting from the smallest symbol (or the equivalently the largest symbol). We assign this symbol to a partition in which it now is the only element. We then consider the next smallest symbol. We now have two choices. The first choice is to create a new partition for this new symbol, and thereby closing the previous partition. The second choice is to include this symbol into the previous partition that contained only the first symbol. We compare the "cost" of these two choices and using a greedy strategy to select the one that has lower cost. We continue in this manner visiting each new distinct symbol and assigning it either to the previous partition or to a new partition of its own.

More generally, as before let us assume that the input sequence consists of N distinct values: $z_1 < z_2 < \cdots < z_N$. Further let's assume our current working partition is $(z_{q_i}, z_{q_j}]$. We now consider the symbol $z_{q_{j+1}}$. We compute the "cost" of the following two choices:

1. Add symbol $z_{q_{j+1}}$ to the partition $(z_{q_i}, z_{q_j}]$ so that our current working partition becomes $(z_{q_i}, z_{q_{j+1}}]$.
2. Terminate our current working partition $(z_{q_i}, z_{q_j}]$ and create a new working partition $(z_{q_j}, z_{q_{j+1}}]$.

The question that arises is how to compute the cost of the two choices we are facing at each step. Essentially we calculate an estimate of the resulting cumulative code size for each choice and select the one that leads to lower code size. To estimate the cumulative code size we use the following expression:

$$C(q_{j+1}) = f(q_1) + f(q_2) + \ldots f(q_j) + g(q_{j+1})$$

where $f(q_i)$ is an estimate of the code length for the partition $(z_{q_{i-1}}, z_{q_i}]$ and is of the form:

$$f(q_i) = n \times (\log p + \log d)$$

where n is the number of values in the partition $(z_{q_{i-1}}, z_{q_i}]$, p is the statistics of the input values falling into this partition, i.e., n/N, and d is the distance between the upper bound and the lower bound in this partition, that is, $z_{q_i} - z_{q_{i-1}}$. The component $g(\cdot)$ is an estimate of the size of the Huffman table, that is,

$$g(q_i) = (\log t + 32) \times t$$

where t is the number of partitions that we would have accumulated thus far after dealing with the current symbol $z_{q_{j+1}}$. Basically, we are assuming that the Huffman table is stored in the same canonical format as in the JPEG standard, namely, by storing codeword lengths, and $\log t$ is an estimate of this. However, in our case we also need to store the symbols up to and including the one where the current partition ends. This requires the additional 32 bits for each entry.

Clearly the above algorithm is sub-optimal. In fact, scanning the symbols from low to high values and from high to low values will yield two different partitioning schemes. In practice, we scan in both directions and select the partitioning scheme that results in the lowest code length.

There are other approaches possible for designing simple heuristics for the alphabet partitioning problem. We experimented with quite a few. For example, a hierarchical scheme, a pivoting scheme, and so on. However, the simple greedy technique described above appeared to give the best performance on the data sets that we tested on. Developing more efficient heuristics is a topic of our ongoing research, and we will report any significant development we may make in the journal version of this paper.

Dataset	Greedy	Dynamic	8-bit	16-bit	32-bit	MH 8-bit	MH 16-bit	Gzip
blunt	286388	275172	531444	337408	1090900	581060	689588	299042
comb	564576	530264	1008068	665372	1436344	654304	824780	678505
post	660972	633096	1274704	887976	2943412	1519556	1873400	760899
delta	2137092	2029452	3189348	2667056	6184832	2813848	3600348	2494699

Table 1: Compression results in bytes with graphics data using different coding schemes.

4 Experimental Results

To evaluate the effectiveness of our algorithms we ran them on a set of data obtained from a computer graphics application. Specifically the input data consisted of the vertex list of a tetrahedral mesh model. Each vertex contained its x, y, z, and scalar values, each represented as a 32-bit floating-point number. Typically these vertex lists are compressed in the graphics literature using some simple modified Huffman coding or using a string compression algorithm like gzip. The four models used and their uncompressed file sizes in numbers of bytes (based on binary 32-bit representation for each component of each vertex) indicated in parenthesis were: `blunt` (655364), `comb` (752404), `post` (1755908), and `delta` (3386884).

To remove first order redundancies in the vertex list we used the most widely employed technique for vertex compression, namely differential coding: store the first entry completely, and then store each remaining entry as its difference from the previous entry. If vertices are close to one another then clearly these differences will be small and can be encoded efficiently. To get better compression we clustered vertices, and within each cluster we formulate and solve a *traveling salesperson problem* (TSP) to get a "good" ordering for differential coding. It should be noted that we followed this procedure prior to all the coding techniques shown in Table 1, except for gzip, where it did not seem to help for obvious reasons.

We then treated each of the x, y, z and scalar values separately, and constructed for each a separate modified Huffman code for the differences by the dynamic programming method described in Section 2 as well as the greedy heuristic given in Section 3. The results are shown in Table 1. It can be seen that the greedy heuristic gives results within 5% of those obtained by dynamic programming. To compare our results we constructed a separate Huffman code for each byte (8 bits), for each pair of bytes (16 bits), and for each four-byte word (32 bits), in each floating-point number, on the same sequence of symbols as encoded by the dynamic programming and the greedy approach. It can be seen that our results are significantly better. Next we used an 8-bit and 16-bit modified Huffman code to the same sequence. Results show that the partitioning scheme we employ yields tremendous benefits over ad-hoc modified Huffman coding. Finally we compared the results with gzip, a popular string compression algorithm and again our results are significantly superior.

All the above results also include the cost of storing the Huffman table. It should be noted that the greedy partitioning approach often resulted in a partition with only

Dataset	Greedy	8-bit	16-bit	32-bit	MH 8-bit	MH 16-bit	Gzip
(0, 100)	1.21	1.51	1.42	1.52	3.12	2.12	1.72
(0, 1000)	1.66	1.88	2.55	3.67	3.42	4.03	2.31
(0, 10000)	2.16	2.33	3.57	6.15	3.42	2.12	2.83

Table 2: Compression results in bytes per number required to encode 10,000 random Gaussian numbers with different coding schemes, where (a, b) means the mean is a and the standard deviation is b.

Dataset	Greedy	8-bit	16-bit	32-bit	MH 8-bit	MH 16-bit	Gzip
(0, 100)	1.17	1.45	1.30	1.20	3.12	2.13	1.69
(0, 1000)	1.62	1.82	1.84	1.93	3.12	2.13	2.21
(0, 10000)	2.07	2.23	2.95	4.15	3.12	2.13	2.79
(0, 100000)	2.57	2.53	3.37	6.52	3.12	2.48	3.18

Table 3: Compression results in bytes per number required to encode 100,000 random Gaussian numbers with different coding schemes, where (a, b) means the mean is a and the standard deviation is b.

one element. To handle this case we included one bit in every entry of the Huffman table to indicate this fact and hence saved the bits to code the rank of this symbol whenever it occurred.

To show the applicability of our approach to a wider class of applications, we then generated random Gaussian numbers with different variances and used the greedy heuristic, Huffman coding of bytes, Huffman coding of pairs of bytes and Huffman coding of four-byte words. Table 2 and Table 3 show the results obtained with a source sequence of length 10,000 and 100,000 respectively. As before all costs including that of storing the Huffman table are included in the results. Also shown are results with 8-bit and 16-bit modified Huffman coding, as well as those with gzip. Again we see that alphabet partitioning based on the greedy heuristic gives superior results.

5 Conclusions

In this paper we formulated the problem of optimal alphabet partitioning for the design of a two-layer semi-adaptive code and gave a solution based on dynamic programming. Since the complexity of the dynamic programming approach can be quite prohibitive for a long sequence and a very large alphabet size, we presented a simple greedy heuristic that has more reasonable complexity. Our experimental results demonstrate that superior prefix coding schemes for large alphabets can be designed using our approach as opposed to the typically ad-hoc partitioning approach applied in the literature.

References

[1] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[2] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40:1098–1101, 1951.

[3] R. Hunter and A. H. Robinson. International digital facsimile standards. *Proceedings of the IEEE*, 68(7):855–865, 1980.

[4] En-Hui Yang; Yunwei Jia. Universal lossless coding of sources with large and unbounded alphabets. In *IEEE International Symposium on Information Theory,*, page 16, 2000.

[5] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

[6] A. Moffat, A.; Turpin. Efficient construction of minimum-redundancy codes for large alphabets. *IEEE Transactions on Information Theory*, 44(4):1650 –1657, 1998.

[7] MPEG. Information technology-generic coding of moving pictures and associated audio information: Video. MPEG: ISO/IEC 13818-2:1996(E), 1989.

[8] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Rostrand Reinhold, 1993.

[9] J. J. Rissanen. Universal coding, information, prediction and estimation. *IEEE Transactions on Information Theory*, 30:629–636, 1984.

[10] A. Said and W.A. Pearlman. Low-complexity waveform coding via alphabet and sample-set partitioning. In *Visual Communications and Image Processing '97, Proc. SPIE Vol. 3024*, pages 25–37, 1997.

[11] S. Smoot and L. Rowe. Study of DCT coefficient distributions. In Proceedings of the SPIE Symposium on Electronic Imaging, Volume 2657, San Jose, CA, January, 1996.

[12] J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of ACM*, 34(4):825–845, 1987.

[13] Bin Zhu, En hui Yang, and Ahmed Tewfik. Arithmetic coding with dual symbol sets and its performance analysis. *IEEE Transactions on Image Processing*, 8:1667 –1676, December 1999.

[14] X. Zou and W.A. Pearlman. Lapped orthogonal transform coding by amplitude and group partitioning. In *Applications of Digital Image Processing XXII, Proceedings of SPIE Vol. 3808*, pages 293–304, 1999.