Alphabet Partitioning Techniques for Semi-Adaptive Huffman Coding of Large Alphabets*

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

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

(Revised March 20, 2006)

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 finding an alphabet partitioning for the design of a two-layer semi-adaptive code as an optimization problem, 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 also give a simple greedy heuristic algorithm whose running time is linear in the length of the input sequence, irrespective of the underlying alphabet size. Although our dynamic programming and greedy algorithms do not provide a globally optimal solution for the alphabet partitioning problem, experimental results demonstrate that superior prefix coding schemes for large

*A preliminary version of this paper appeared in *Proc. IEEE Data Compression Conference (DCC '03)*, pp. 372-381, March 2003.

†dchen@cis.poly.edu. Research supported by NSF Grant CCF-0118915.

‡yjc@poly.edu. Research supported in part by NSF Grant CCF-0118915, NSF CAREER Grant CCF-0093373, and NSF Grant CCF-0541255.

§memon@poly.edu. Research supported in part by NSF Grant CCF-0118915 and NSF Grant CCF-0208678.

¶xwu@poly.edu. Research supported in part by NSF Grant CCF-0208678.

alphabets can be designed using our new approach.

Keywords: Data compression, two-layer semi-adaptive coding, large alphabet partitioning, dynamic programming, greedy heuristic.

1 Introduction

For various reasons, some of them technical and some not, the most widely used entropy coding technique today is Huffman coding [5]. Although adaptive versions of Huffman's algorithm have been known for many years [8, 18], primarily 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 which makes two passes through the data is used. Here, in the first pass, statistics are collected to aid the construction of the optimal Huffman code for the given instance of the source sequence. The actual coding is then done in the second pass. Since the Huffman code arrived at is data specific, it needs to be included in the data using some canonical representation technique. Widely used compression standards like JPEG, GIF, gzip, etc. include a semi-adaptive Huffman code as one of the supported, and indeed most commonly employed, 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 starts 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 in the Huffman table. An excessively large Huffman table can lead to multiple problems. First of all, in the semi-adaptive case, a larger table would require more bits to represent, thereby reducing the efficiency of Huffman coding. For example, symbols that occur only once need to be explicitly or implicitly specified along with their Huffman code. Although the cost for this can be significantly reduced for dense alphabets [17], it can still be prohibitively high for sparse alphabets and could even result in data expansion. Secondly, in both the static and semi-adaptive case, large Huffman tables can lead to

serious difficulties in a hardware implementation of the codec. This is due to the fact that, given its large size, the Huffman table may need to be stored off-chip, leading to the CPU-Memory bandwidth problem which is a well known bottleneck in modern computing architectures. For special cases like probability sorted alphabets, clever techniques are known that require significantly less memory, but in general, Huffman table size is a serious issue for hardware implementations.

1.1 Alphabet Partitioning

One way to deal with the problem of entropy coding of 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* [15] 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 [13] 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 multiple 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 [6] and the MPEG video compression standards [12] 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 when encoding floating-point numerical data that arise in compression of scientific applications. 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 [21].

1.2 Our Contribution - Dynamic Programming and Greedy Algorithms for 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 [16]. 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 $\mathcal{Z}$ 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 value within the partition? By optimal here, we mean a code that minimizes the number of bits needed to encode the source sequence plus the number of overhead bits needed to describe the code.

In this paper, we present an $O(N^3)$-time dynamic programming algorithm and an $O(N)$-time greedy heuristic method for partitioning an alphabet with the goal that the overall combined cost of representing the message and the Huffman table is minimized, where N is the number of distinct symbols being encoded. In both algorithms, an additional time linear in the length of the input message is needed to read the message and collect the symbol frequencies, prior to performing alphabet partitioning. It turns out that we do not optimally solve the alphabet partitioning problem. This is due to the fact that it is difficult to accurately formulate the cost of the final Huffman table in our dynamic programming formulation. Even though the resulting dynamic programming solution provides an optimal solution to our cost function,

because the cost function itself does not *exactly* capture the cost of the final Huffman table, the solution does not achieve the real optimal. Our alternative algorithm, the greedy approach, is a heuristic method with local considerations and hence does not achieve the global optimal either (but is much faster than dynamic programming). We pose the question of how to (provably) solve the alphabet partitioning problem optimally for semi-adaptive modified Huffman code as an open question. Although this problem is not solved optimally, our problem formulation and the developed solutions, the dynamic programming and the greedy algorithms, result in superior overall coding efficiencies in our semi-adaptive modified Huffman code, as demonstrated in the experimental results.

The rest of this paper is organized as follows. In the next section we formulate the problem of finding a good alphabet partitioning for the design of a two-layer semi-adaptive entropy code as an optimization problem, 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 linear in the length of the input sequence. 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 methods applied in the literature.

1.3 Related Work

The problem of a 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 [2] 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.* [20], 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.

Itoh [7] has given a universal noiseless coding technique for large alphabets where the per-letter redundancy is bounded by a factor that does not include alphabet size. As discussed in Effros et al. [3], in Itoh's approach the encoder first sends to the decoder a model of the distribution of the data, and then describes the data using this model. The encoder chooses the model that minimizes the total description length of the data, i.e., the length of the model description plus the length of the data description given the model. We refer to [3, 7] for more details.

The multi-level coding scheme has been applied very effectively in the Burrows-Wheeler Transform (BWT) compression; see the work of Fenwick [4] for an example. More recently Yang and Jia [19] 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 [11] and the references therein.

Finally and most importantly, Liddell and Moffat [9], independent of this work and roughly around the same time as this work was done, considered the problem of encoding a large alphabet source sequence by using a two-layer code called a *K-flat* code. The structure of a K-flat code can be represented by a tree, which has a binary upper section of $K = 2^k$ nodes for some integer k, and each of the nodes at depth k is the root of a strictly binary subtree where all leaves are at the same depth. Therefore, the first-layer code has exactly k bits that acts as a *subtree selector* for picking one of the K binary subtrees. They formulate the problem of finding an optimal partitioning scheme as a dynamic programming problem and give an $O(KN \log N)$-time solution by exploiting some structure in the optimization problem [9, 10]. They assume that the symbols in the alphabet are sorted by the frequency values, and hence in that sense their method is less general than our approach here. Moreover, their first-layer code is a fixed-length binary code of k bits, as opposed to a Huffman code in our case, and thus their compression efficiency is in general not expected to be as good as ours. However, the advantage of their K-flat code is its structural simplicity: each codeword consists of a k-bit prefix and a variable-length suffix whose length is

completely defined by the prefix; this allows a simple decoding process. Therefore, the main strength of their coding scheme is fast decoding rather than compression efficiency.

2 Alphabet Partitioning Using Dynamic Programming

Our problem is to code a finite input sequence S drawn from a very large alphabet $\mathcal{Z}$ 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 Length (MDL) principle [14].

We assume that the alphabet $\mathcal{Z}$ is ordered, which is very large. Suppose that the input sequence consists of N distinct values: $z_1 < z_2 < \cdots < z_N$, and that U is the total range of values any input symbol could fall into (U is decided by the given input data representation—e.g., for input symbols given as 4-byte integers, $U = 2^{32}$, and $\log_2 U = 32$ bits suffice to represent any input symbol). Central to our two-layer code design is the formulation of alphabet partitioning as an optimization problem. 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: $S(q_{i-1}, q_i] = \{z_j | j \in (q_{i-1}, q_i]\}$, $1 \leq i \leq M$, 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

$$H(q_1, q_2, \cdots, q_{M-1}) = -\sum_{i=1}^{M} P(S(q_{i-1}, q_i]) \log_2 P(S(q_{i-1}, q_i]). \tag{1}$$

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

We still need to resolve the remaining uncertainty for the symbols within a cell of the alphabet partition, which is the second layer of our code. We use a fixed-length code for this second-layer coding, as is done by the JPEG modified Huffman code.

Optionally, we could use a variable-length code (such as Golomb code) to probably further improve the compression efficiency. However, using a fixed-length code has an advantage of being simple and yet still quite efficient, as demonstrated by our experiments.

The proposed two-layer coding scheme is semi-adaptive. The encoder first scans the input sequence and collects the statistics to find a good alphabet partition as described by the following dynamic programming algorithm. It then sends side information about the alphabet partition, namely the values $z_{q_1}, z_{q_2}, \cdots, z_{q_{M-1}}$ that define the partition. Note that scanning the input sequence and collecting the statistics takes time linear in the length of the input sequence (e.g., by using a hash table), prior to performing alphabet partitioning.

Now we present a dynamic programming algorithm for our alphabet partitioning. Let $L(z_j|j \in (q_{i-1}, q_i])$ be the length of the second-layer fixed-length code of the set $S(q_{i-1}, q_i] = \{z_j|j \in (q_{i-1}, q_i]\}$. Obviously, we have $L(z_j|j \in (q_{i-1}, q_i]) = |S(q_{i-1}, q_i]| \log_2(q_i - q_{i-1})$. Let $R_k(0, n]$ be the overall code length produced by the two-layer coding scheme under the best k-cell alphabet partition that we can achieve for the subset $\{z|z < z_n\}$ of the input data set. Namely,

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

Here $T(q_1, q_2, \cdots, q_{k-1})$ is the Huffman table size of the first-layer code for the k-cell partition by $(q_1, q_2, \cdots, q_{k-1})$ on the subset $\{z|z < z_n\}$. Such Huffman table size depends on how the table is represented. Assuming that it is stored in the same canonical format as in the JPEG standard, i.e., by storing the codeword lengths, then the table size is upper-bounded by $k \log_2 k$ bits. In our case we also need to store the information on the partition. This requires an additional $\log_2 U$ bits for each entry. Hence $k(\log_2 k + \log_2 U)$ is an upper bound for $T(q_1, q_2, \cdots, q_{k-1})$. We remark that for canonical Huffman table, its sub-table size $T(q_1, q_2, \cdots, q_{k-1})$ corresponding to the sub-problem $\{z|z < z_n\}$ in intermediate steps of dynamic programming cannot be expressed exactly, therefore we let $T(q_1, q_2, \cdots, q_{k-1}) = k(\log_2 k + \log_2 U)$, which is only an *estimation*. Because $T(\cdot)$ is only an estimation, our resulting dynamic programming solution is not optimal, albeit it is optimal with respect to the cost function we use in the dynamic programming formulation. For other representations of the

Huffman table in the first-layer code, $T(\cdot)$ needs to be appropriately modified, and again the resulting dynamic programming solution is not optimal if the corresponding $T(\cdot)$ is not expressed exactly.

With $R_k(0,n]$ given above, 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 procedure is based on the recursion:

$$\begin{aligned} R_k(0,n] &= \min_{0<q<n} \{R_{k-1}(0,q] + L(z_j|j \in (q,n]) - |S(q,n]| \log_2 P(S(q,n])\} \\ &\quad -(k-1)\log_2(k-1) + k\log_2 k + \log_2 U. \end{aligned} \tag{4}$$

After pre-computing

$$F(a,b] = L(z_j|j \in (a,b]) - |S(a,b]| \log_2 P(S(a,b]), \quad 0 < a < b \leq N \tag{5}$$

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

```
/* Initialization */
q_0 ≡ 0;
q_M ≡ N;
for n := 1 to N do
    R_1(0,n] := L(z_j|j ∈ (0,n]) + log_2 U;
/* Minimize the description length */
for m := 2 to N do
    for n := m to N do
        R_m(0,n] := min_{0<q<n}{R_{m-1}(0,q] + F(q,n]} - (m-1) log_2(m-1) + m log_2 m
                    + log_2 U;
        Q_m(n) := arg min_{0<q<n}{R_{m-1}(0,q] + F(q,n]};
/* Construct the alphabet partition */
M := arg min_{1≤m≤N} R_m(0,N];
for m := M down to 2 do
    q_{m-1} := Q_m(q_m);
output final M-cell alphabet partition (q_0, q_1, ··· , q_M).
```

The complexity of the above dynamic programming algorithm is clearly $O(N^3)$. Recall that we also spend an additional time linear in the length of the input sequence to collect the symbol frequencies before performing the dynamic programming algorithm. 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 a fixed-length code for encoding the value of a symbol within a partition. However, we could alternatively use a variable-length code (such as Golomb code) for this purpose and the above algorithm can be easily modified to yield a suitable partitioning of the alphabet. We remark that, as mentioned before, using a fixed-length code in the second layer already compresses quite well as shown by our experiments, and we suspect that the additional complexity of using a variable-length code in the second layer may not pay off in practice.

3 Greedy Heuristic

The running time of the dynamic programming algorithm as given in the previous section is $O(N^3)$. 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 greedy heuristic for partitioning the alphabet, which runs in time linear in the number of distinct symbols in the source sequence, that is, in $O(N)$ time. Prior to performing alphabet partitioning, the encoder needs an additional time linear in the length of the input sequence to collect the symbol frequencies, as is the case described in the previous section. In the next section we give experimental results showing that the greedy heuristic yields results quite close to those obtained by the dynamic programming approach, and hence can be an attractive alternative in many applications.

As in the previous section, we assume that the distinct symbols that occur in the source sequence are ordered from the smallest to the largest. The greedy algorithm works as follows. We make a linear scan of the symbols starting from the smallest symbol (or 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 have two choices. The first choice is to create a new partition for this new symbol, 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 use 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 we assume that the input sequence consists of N distinct values: $z_1 < z_2 < \cdots < z_N$ and that $\log_2 U$ bits suffice to represent any input symbol, where U is the total range of values any input symbol could fall into. Suppose 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 estimation of the resulting cumulative code size for each choice and select the one that leads to lower code size. In general, to estimate the cumulative code size $C(q_k)$ right after dealing with the symbol z_{q_k} we use the following expression:

$$C(q_k) = f(q_1) + f(q_2) + \ldots + f(q_k) + g(q_k), \tag{6}$$

where $f(q_k)$ is an estimate of the code length for the partition $(z_{q_{k-1}}, z_{q_k}]$. Note that this partition is equivalent to the partition $[z_{(q_{k-1}+1)}, z_{q_k}]$ as there is no input symbol falling into the interval $(z_{q_{k-1}}, z_{(q_{k-1}+1)})$ to be coded. We express $f(q_k)$ as

$$f(q_k) = n \cdot (-\log_2 p + \log_2 d). \tag{7}$$

Here n is the number of distinct input symbols in the partition $[z_{(q_{k-1}+1)}, z_{q_k}]$, and p is the empirical probability of the input values falling into this partition, i.e., $\sum_{(q_{k-1}+1) \leq r \leq q_k} w_r/W$, where w_r is the number of times the input symbol z_r appears in the input sequence and W is the sum of all such w_r's. Finally, d is the distance between the upper bound and the lower bound of this partition, that is, $z_{q_k} - z_{(q_{k-1}+1)}$. The component $g(\cdot)$ is an estimate of the size of the Huffman table, that is,

$$g(q_k) = t \cdot (\log_2 t + \log_2 U), \tag{8}$$

where t is the number of partitions that we have accumulated thus far right after dealing with the symbol z_{q_k}. As before, 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 $t \log_2 t$ is an estimate of this. Also, in our case we need to store the information on the partition. This requires the additional $\log_2 U$ bits for each entry. It is easy to see that for the current symbol $z_{(q_j+1)}$, $C(q_j + 1)$ has two possible values corresponding to the two choices of where to place $z_{(q_j+1)}$.

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. Typically, the gap between the code lengths of the two schemes is very small, indicating that the method is reasonably stable. In practice, we scan in both directions and select the partitioning scheme that results in the lowest code length.

There are other possible approaches 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 all the datasets that we have tested.

4 Experimental Results

To evaluate the effectiveness of our algorithms, we implemented in C/C++ the dynamic programming method described in Section 2 and the greedy heuristic given in Section 3, where in both methods we used a Huffman code for the first-layer coding and a fixed-length code for the second-layer coding. All the experiments were conducted on a Sun Blade 1000 workstation with dual 750MHz UltraSPARC III CPUs and 4GB of main memory, running under the Solaris operating system.

We present the experimental results on real datasets in Section 4.1 and those on synthetic datasets in Section 4.2; additional experiments on a family of datasets to test how close to optimal our dynamic programming and greedy heuristic achieve are reported in Section 4.1 and Appendix A of the technical report of this paper [1].

4.1 Experiments on Real Datasets

To evaluate the effectiveness of our algorithms on real datasets, we ran them on a set of data obtained from a computer graphics application. Specifically, each input dataset contains a vertex list of a triangle-mesh model. The four datasets used were:

`bunny`, `bunny2`, `phone` and `phone2`. The `bunny` and `bunny2` datasets represent the same model of a bunny but the vertices are sampled at different resolutions, where each vertex contains five components: its x, y, z, scalar, and confidence values, each represented as a 32-bit floating-point number. Similarly, the `phone` and `phone2` datasets represent the same model of a phone but the vertices are sampled at different resolutions, where each vertex contains only three components: its x, y, and z values, again 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.

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. We treated each of the five or three components (x, y, z, scalar and confidence components for the bunny models, and x, y, and z components for the phone models) *separately*, and constructed for each a separate modified Huffman code for the differences by the dynamic programming method as well as the greedy heuristic. To compare our results we also 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 (i.e., obtained by taking differences) as encoded by the dynamic programming and the greedy approaches. In addition, we used an 8-bit and 16-bit modified Huffman code (respectively denoted by MH-8 and MH-16 in all tables below) to the *same sequence*, as well as using gzip, a popular string compression algorithm. We implemented all these coding methods in C/C++ except for gzip, for which we used the command "gzip" provided in Unix. As mentioned, all these coding methods were applied to the *same sequence* for fair comparisons. For gzip, we always used the option of the best compression ("gzip -9"). In addition, we also computed the 32-bit entropy (entropy of the 32-bit symbols) of the same sequence of symbols.

First we show in Table 1 the compression results on the `bunny` dataset. Since the dynamic programming method is time consuming, we only experimented on compressing the first 10,000 entries for each of the x, y, z, scalar and confidence components. For each coding scheme involving Huffman code, we list the cost of storing the Huffman table, the cost of storing the data, and the total cost of storing both, for each compression result. Looking at the total costs of storing both table and data, we

bunny	Dyn.	Greedy	8-bit	16-bit	32-bit	MH-8	MH-16	Gzip	Entropy
x-table	272	672	1108	23921	41148	21	181		
x-data	23128	23600	24932	20211	15100	32275	28063		
x-all	23400	24272	26040	44132	56248	32296	28244	25525	15930
y-table	90	41	38	40	37964	38	52		
y-data	4838	4879	8594	6092	18512	34718	26108		
y-all	4928	4920	8632	6132	56476	34756	26160	6031	4831
z-table	102	695	1052	14925	41103	16	39		
z-data	18762	17773	20612	16167	17377	31256	24729		
z-all	18864	18468	21664	31092	58480	31272	24768	30104	13891
scalar-table	85	755	1060	28541	41260	19	54		
scalar-data	22407	22837	24132	18887	16280	31967	26150		
scalar-all	22492	23592	25192	47428	57540	31986	26204	32086	16472
conf-table	126	209	1052	12191	22602	19	46		
conf-data	10858	11123	13980	9185	13550	31525	25394		
conf-all	10984	11332	15032	21376	36152	31544	25440	8456	7537
Total-table	675	2372	4310	79618	184077	113	372		
Total-data	79993	80212	92250	70542	80819	161741	130444		
Total-all	80668	82584	96560	150160	264896	161854	130816	102202	58661

Table 1: Compression results in bytes with the `bunny` graphics dataset using different coding schemes. For each of the x, y, z, scalar and confidence components, we show the results of compressing the first 10,000 entries (shown as a breakdown between the cost of the Huffman table ("-table") and the cost of coding the data ("-data"), and the total cost ("-all")), as well as the corresponding 32-bit entropy. The original data size is 40,004 bytes for each such component (a 4-byte float for each of the 10,000 entries, plus a 4-byte integer to indicate the number of entries). "Dyn." denotes dynamic programming.

Dataset	Greedy	8-bit	16-bit	32-bit
bunny-table	8,864	5,230	256,914	736,734
bunny-data	328,576	342,038	307,090	340,626
bunny-all	337,440	347,268	564,004	1,077,360
bunny2-table	36,957	5,294	377,940	2,173,851
bunny2-data	1,012,619	1,138,758	949,532	1,825,109
bunny2-all	1,049,576	1,144,052	1,327,472	3,998,960
phone-table	11,291	2,750	274,508	1,020,159
phone-data	446,329	501,698	407,744	510,953
phone-all	457,620	504,448	682,252	1,531,112
phone2-table	19,042	2,750	180,626	3,081,475
phone2-data	1,294,286	1,517,202	1,219,862	1,737,901
phone2-all	1,313,328	1,519,952	1,400,488	4,819,376

Dataset	MH-8	MH-16	Gzip	Entropy
bunny-table	178	16,850		
bunny-data	611,390	587,262		
bunny-all	611,568	604,112	585,242	235,962
bunny2-table	202	20,891		
bunny2-data	1,793,014	1,690,293		
bunny2-all	1,793,216	1,711,184	1,719,242	762,300
phone-table	118	11,846		
phone-data	867,702	821,634		
phone-all	867,820	833,480	895,969	342,129
phone2-table	141	13,778		
phone2-data	2,602,739	2,415,334		
phone2-all	2,602,880	2,429,112	2,584,875	983,496

Table 2: Compression results with the graphics datasets using different coding schemes. For each dataset, we show the compressed size in bytes for the *entire* vertex list, as well as the corresponding 32-bit entropy in bytes. For each coding scheme involving Huffman code, we list the cost of storing the Huffman table ("-table"), the cost of storing the data ("-data"), and the total cost ("-all"), for each compression result. The original vertex-list sizes in bytes (and numbers of vertices) are as follows: `bunny`: 718,948 (35,947), `bunny2`: 2,107,968 (105,398), `phone`: 996,536 (83,044), and `phone2`: 2,988,092 (249,007).

see that the greedy heuristic gives results very close to those obtained by dynamic programming—at most 4.9% worse. Surprisingly, in some rare cases such as the y and z components, the greedy heuristic gives a slightly better result than dynamic programming. This is counter-intuitive, since dynamic programming is a global optimization technique whereas the greedy heuristic is only a local optimization method. However, as mentioned in Section 2, the cost function involved in the dynamic programming formulation, in particular, the cost $T(\cdot)$ that represents the Huffman table size, is only an estimation and thus does not always reflect the actual cost precisely. Because of this, the greedy approach can give better results in some rare cases, although in most cases the dynamic programming results are better. It should be noted that the dynamic programming method still gives the optimal solution with respect to the given cost function. We remark that the 32-bit entropy is always a lower bound for the compression results of all approaches compared, as expected.

Comparing our results of dynamic programming and of greedy method with those of 8-bit, 16-bit, and 32-bit Huffman code in Table 1, we see that our results are significantly better. In particular, our Huffman-table costs are significantly smaller, and our data costs are typically smaller or comparable, resulting in great improvements in overall compression efficiency of our methods. Comparing our results with those of 8-bit and 16-bit modified Huffman code, we see that the latter two methods, though have very small Huffman-table costs, have significantly larger data costs. Overall, the partitioning scheme we employ yields tremendous benefits over the ad-hoc modified Huffman coding approaches. Finally, compared with the results of gzip, again our results are significantly superior. We remark that gzip performs extremely well on the confidence component. This is because there are many identical values in the confidence-component sequence; gzip can concatenate many of them into a single code word and represent such repeated patterns easily. Therefore gzip can outperform our two-layer coding in this particular case. Except for this special case, our results are significantly superior to those of gzip in all other cases, including the overall compression results combining all components together.

For the entries in Table 1, we also measured the corresponding running times. The encoding speeds of all the compression methods listed are fast and roughly the same, except for dynamic programming, which is extremely slow. For example, it took more than 23 hours for dynamic programming to encode the first 10,000 entries of one component, for each of the five components, with a total running time of more

than 115 hours. For the greedy method, on the other hand, the encoding times for the first 10,000 entries of one component range from 0.06 second to 0.1 second, with a total running time of 0.4 second. The decoding speeds of all methods are similar and fast—they are slightly faster than the encoding speed of the greedy approach. Clearly, for large datasets, the greedy heuristic is much more favorable than the dynamic programming method.

Since the greedy approach runs much faster than the dynamic programming method and gives compression results almost as good, in the remaining experiments we only ran the greedy heuristic as our representative results. In Table 2, we show the results of compressing all four graphics datasets, where for each dataset we compressed the *entire* vertex list (as opposed to the first 10,000 vertices in Table 1). Similar to what we observed in Table 1, our partitioning scheme based on the greedy method results in significant advantages over all other methods being compared.

It should be noted that the dynamic programming and the greedy partitioning approaches often resulted in a partition with only 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 value of this symbol whenever it occurred.

4.2 Experiments on Synthetic Datasets

To show the applicability of our approaches to a wider class of applications, we also generated random Gaussian numbers with different variances and used the greedy heuristic, Huffman coding of bytes (8 bits), Huffman coding of pairs of bytes (16 bits) and Huffman coding of four-byte words (32 bits), as well as 8-bit and 16-bit modified Huffman coding and gzip. As before, we used the option of the best compression ("gzip -9") for gzip, and we also computed the 32-bit entropy. Table 3 and Table 4 show the results obtained with a source sequence of length 10,000 and 100,000 respectively. As before each result is shown as the cost of storing the Huffman table, the cost of storing the data, and the total cost of storing both, for each coding scheme involving Huffman code. Again we see that alphabet partitioning based on the greedy heuristic gives superior results.

Dataset	Greedy	8-bit	16-bit	32-bit	MH-8	MH-16	Gzip	Entropy
(0,100)-table	0.403	0.42	1.91	32.95	0.0128	0.0144		
(0,100)-data	9.280	11.65	10.16	12.66	25.00	17.01		
(0,100)-all	9.683	12.07	12.07	45.61	25.013	17.024	13.28	9.12
(0,1000)-table	0.529	0.48	11.10	33.00	0.0128	0.0144		
(0,1000)-data	12.674	14.54	12.97	13.64	25.00	17.01		
(0,1000)-all	13.203	15.02	24.07	46.64	25.013	17.024	18.16	11.92
(0,10000)-table	0.574	0.83	21.78	33.01	0.0128	0.0144		
(0,10000)-data	16.738	17.78	14.18	13.07	25.00	17.01		
(0,10000)-all	17.312	18.61	35.96	46.08	25.013	17.024	22.64	13.1

Table 3: Compression results in bits 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. For each coding scheme involving Huffman code, we list the cost of storing the Huffman table ("-table"), the cost of storing the data ("-data"), and the total cost ("-all"), for each compression result. We also show the corresponding 32-bit entropy in bits per number.

5 Conclusions

In this paper we formulated the problem of finding a good alphabet partitioning for the design of a two-layer semi-adaptive code as an optimization problem, 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 also presented a simple greedy heuristic that has more reasonable complexity. Our experimental results demonstrated that superior prefix coding schemes for large alphabets can be designed using our approach, as opposed to the typically ad-hoc partitioning methods applied in the literature.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions that greatly improved this paper.

References

[1] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Alphabet partitioning techniques for semi-adaptive Huffman coding of large alphabets. Technical Report TR-

Dataset	Greedy	8-bit	16-bit	MH-8	MH-16	Gzip	Entropy
(0,100)-table	0.102	0.04	0.24	0.00128	0.00144		
(0,100)-data	9.254	11.64	10.21	25.00	17.00		
(0,100)-all	9.356	11.68	10.45	25.001	17.001	12.64	9.17
(0,1000)-table	0.386	0.05	1.88	0.00128	0.00144		
(0,1000)-data	12.548	14.54	13.47	25.00	17.00		
(0,1000)-all	12.934	14.59	15.35	25.001	17.001	17.04	12.44
(0,10000)-table	0.510	0.08	10.94	0.00128	0.00144		
(0,10000)-data	16.035	17.78	16.28	25.00	17.00		
(0,10000)-all	16.545	17.86	27.22	25.001	17.001	22.24	15.24
(0,100000)-table	0.544	0.09	12.32	0.00128	0.00472		
(0,100000)-data	19.986	20.23	18.74	25.00	19.89		
(0,100000)-all	20.530	20.32	31.06	25.001	19.895	25.44	16.41

Table 4: Compression results in bits 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. For each coding scheme involving Huffman code, we list the cost of storing the Huffman table ("-table"), the cost of storing the data ("-data"), and the total cost ("-all"), for each compression result. We also show the corresponding 32-bit entropy in bits per number. We omit the entries for "32-bit" since these results are much worse than all other methods, similar to what we have seen in Table 3.

CIS-2006-02, Department of Computer and Information Science, Polytechnic University, 2006.

[2] 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.

[3] M. Effros, P. A. Chou, E. A. Riskin, and R. M. Gray. A progressive universal noiseless coder. *IEEE Transactions on Information Theory*, 40(1):108–117, 1994.

[4] P. Fenwick. The Burrows-Wheeler transform for block sorting text compression: Principles and improvements. *The Computer Journal*, 39(9):731–740, 1996.

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

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

[7] S. Itoh. A source model for universal quantization and coding. In *Proc. Symp. Inform. Theory and Its Appl.*, pages 611–616, 1987 (in Japanese).

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

[9] M. Liddell and A. Moffat. Hybrid prefix codes for practical use. In *Proc. IEEE Data Compression Conference*, pages 392–401, 2003.

[10] A. Moffat. Personal communications, March 2005.

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

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

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

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

[15] 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.

[16] 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.

[17] A. Turpin and A. Moffat. Housekeeping for prefix coding. *IEEE Transactions on Communications*, 48(4):622–628, 2000.

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

[19] E.-H. Yang and Y. Jia. Universal lossless coding of sources with large and unbounded alphabets. In *IEEE International Symposium on Information Theory,*, page 16, 2000.

[20] B. Zhu, E.-H. Yang, and A. Tewfik. Arithmetic coding with dual symbol sets and its performance analysis. *IEEE Transactions on Image Processing*, 8:1667 –1676, December 1999.

[21] 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.