A New Data Compression Method with Specific Application in Bulk Compression

Alessandro Palau and Gagan Mirchandani

Department of Computer Science and Electrical Engineering
University of Vermont, Burlington, Vermont 05405

ABSTRACT

Given a source alphabet of \( M \) symbols and associated probabilities or their estimates, we find a sub-optimal set of codewords using a simple prefix property type iterative algorithm to generate codewords lengths and a look-up table based mapping algorithm for assigning codewords. The expected codeword length \( L_f \) is slightly longer than that obtained for a Huffman code but may also equal it. When equal, the algorithm generates a larger set of applicable codewords. The time complexity for generating lengths and the associated codewords is less than that with Huffman, where these tasks have a complexity of \( O(M \log M) \), while they are of order \( O(M) \) in the new algorithm. For bulk compression, where it is necessary to compress a large number of small files, the algorithm typically shows greater compression efficiency than that obtained with other standard UNIX based compressors.

Index terms — Lossless coding, Huffman coding, text compression, extended alphabet.

1. INTRODUCTION

A simple but effective prefix property type iterative algorithm is developed for encoding data bases with finite sized alphabets. For a given data set, we determine the inter-symbol dependencies by evaluating joint probabilities \( p(i_1, i_2, ..., i_n) \) for various values of \( n \), i.e. the relative frequencies of the \( n \)-tuple event \( i_1, i_2, i_3, ..., i_n \) where the indices \( i \) range over all possible symbols. Since this can lead to large sized alphabets,
we implement the statistical model in a fashion that controls the build up of complexity. Symbols may be sorted in any order and codeword lengths are found iteratively without the constraint of minimization of average codeword length. After ordering codeword lengths, unique codewords are identified with them, using a look-up table based mapping algorithm. Relevant information about the code is stored either in the header of the compressed file or when encoding multiple files, written to a separate file, precluding the necessity of a header in each of the multiple compacted files. Alphanumeric data such as text files, where the source consists of many articles as in an encyclopedia, or very large data bases such as directories containing thousands of small source files, appear ideal candidates for this method. The latter application, hereafter, is referred to as bulk compression. The performance of the algorithm is compared with standard UNIX based compressors compress, gzip and also with other programs yabba and uhap.

1.1. Related work

Because of its importance as a component in data compression schemes, Huffman coding and its variations have been studied extensively over the last two decades. Given a source with alphabet probabilities \( p(1), p(2), \ldots, p(M) \), we determine lengths \( l(1), l(2), \ldots, l(M) \) respectively and the average codeword length \( L = \sum_{i=1}^{M} p(i)l(i) \) subject to the Kraft inequality \( \sum_{i=1}^{M} 2^{-l(i)} \leq 1 \). Without additional constraints, we have the Huffman coding problem. For alphabetic codes we impose the additional requirement to the Huffman problem that the alphabetic order of the source symbols be preserved by the codewords. For monotonic codes we impose the requirement, additional to the alphabetic code requirement, that \( l(1) \leq l(2) \cdots \leq l(M) \) hold. The alphabetic ordering constraint is actually seen to follow from the monotonic ordering property. Thus the monotonic coding problem is equivalent to the Huffman coding problem together with the single additional constraint \( l(1) \leq l(2) \cdots \leq l(M) \).

Time complexity and average codeword length issues in Huffman type problems have been studied by many authors. Voorheis [1] has examined the monotonic coding problem and shown that monotonic codewords can be calculated in \( O(M^2) \) steps. Abrahams [2] has investigated monotonic alphabetic codes and determined bounds on values of \( L_a \) in terms of \( L_h \) and \( L_m \), the optimal alphabetic, Huffman and monotonic average code word lengths respectively: \( L_h \leq L_a \leq L_m \). Hu and Tucker [3] have devised an \( O(M \log M) \) algorithm for optimal alphabetic code. Nakatsu [4] has studied efficiency of alphabetic codes versus Huffman codes.
and showed that $L_a < L_b + 1$. We show here that when the number of distinct probabilities is large, the expected value of the average codeword length $E[L_f] = L_b + \epsilon/2$ where $L_f$ is the average codeword length using the new algorithm, called *Length-first-coding*, abbreviated as $Lfc$, and $\epsilon$ is a parameter. Here $L_f$ is assumed to be a slowly-varying random variable, so it can be considered constant in a small interval $\epsilon$.

The time complexity for finding codeword lengths is shown to be $O(M) \times n$ where $n$, which is independent of $M$ but a function of $\epsilon$, is the number of iterations required for convergence. The cost of ordering symbols in nondecreasing length is again $O(M)$; and finally the cost of assigning codewords is $cM$ where $c$ is a small constant.

Organization of the work is as follows: Section 2 describes the iterative algorithm for determining codeword lengths based on a given statistical model of the data. We show how codeword lengths are determined and formulate bounds on $L_f$. The method for assigning codewords with the prefix property is described in Section 3. Time complexity of the algorithm is established in Section 4. Methods for defining the dictionary and generating successive refinements of the statistical model, for any given data base, are described in Section 5. In Section 6 we describe the application to bulk compression and also to an application where new files or edited files may be compressed without regenerating the alphabet and its statistics. Benchmarks on the performance of the algorithm and other UNIX based compressors in both these applications are given.

## 2. DETERMINING CODEWORD LENGTHS

For a given data set, an alphabet is constructed consisting of a select sequence of patterns within the data set (Section 4). The probability $p_k$ of letter $a_k$, for each symbol in the alphabet $a_1, a_2, \ldots, a_M$, is estimated as the ratio of its occurrence to the total number of symbols in the data. For determining codeword lengths, symbols do not need to be sorted monotonically according to $p_k$.

Codeword lengths for the alphabet symbols $a_k$, $k \in [1 \ldots M]$ are determined as per algorithm A. Algorithm B is invoked if further optimization is desired (Figure 1). As opposed to the design of codewords from the leaves of a tree to the root, we generate an iterative procedure where we first determine lengths and then the unique codewords.
Algorithm A

A1: Set $\theta \leftarrow 0$

A2: Determine $L_k = \lfloor \log_2 \frac{1}{p_k} + \theta \rfloor$, $k \in [1...M]$

A3: Compute $S = \sum_{k=1}^{M} 2^{-L_k}$

A4: If $S > 1$, let $\theta = \theta + \epsilon$ where $0 < \epsilon < 1$ and go to A2.

A5: Else determine if $\gamma = |S - 1| \leq \delta$ for some $\delta > 0$.

A6: If $\gamma \leq \delta$ store $L_k$'s and terminate algorithm.

Algorithm B

B1. If $\gamma > \delta$ decrement $L_m$, the maximum codeword length, by 1, and increment $S$.

B2. If $\gamma = |S - 1| < \delta$ terminate algorithm. Else go to B1.

If $S > 1$, $\theta$ is incremented by $\epsilon$ and steps A2 and A3 are repeated; otherwise all the $L_k$'s are kept as valid ones. Clearly, if $\theta = 0$ and lengths $L_k$ are determined without the floor operation, $S$ would equal 1. Utilizing the floor operation serves to truncate the real numbers in the argument to integers, allowing for an integral number of bits for the lengths. Therefore, in most cases $S > 1$ initially, and steps A2 and A3 need to be iterated until the desired convergence condition is reached. The necessity for $S \leq 1$ is seen as follows. Consider a binary tree of maximal depth $L_m$. The total number of leaves is $2^{L_m}$ and codewords with length $L_k$ cover $2^{L_m-L_k}$ leaves. For a prefix code it is clear that the total number of leaves covered by the codewords cannot exceed the total number of leaves in the tree. Accordingly, we have $\sum_{k=1}^{M} 2^{L_m-L_k} \leq 2^{L_m}$ which gives the Kraft inequality $S \leq 1$ for prefix condition codes.

Empirical results show that the larger the number $N_p$ of distinct alphabet probabilities $p_k$, the closer is $S$ to 1 after convergence. Instead, for the case of many non-distinct probability values, $S$ remains constant for a few iterations, and then jumps to a value much lower than 1. This could be corrected by making the step size $\epsilon$ a function of the symbol index $k$ at step A4. In any event, large data sets with many distinct $p_k$'s generally allows a final value of $S$ close to 1. For situations where $S$ is still not close enough to 1 after convergence, codeword lengths can be further adjusted as per algorithm B. i) Shorten by 1 codewords of length $L_m$, the maximum codeword length, one by one; at each change adjust $S$ by adding $2^{-L_m-1} - 2^{-L_m} = 2^{-L_m}$ to it. Do so until $|S - 1| \leq \delta$. ii) If $S$ is still far from 1, do the same for codewords of length $(L_m - 1)$ and so forth.
2.1. Coding Efficiency: Bounds on $L_f$

We show here that the expected value of the average codeword length, $E[L_f]$ is approximately $L_h + \epsilon/2$. The proof is offered in two parts. We first show that between successive iterations, the expected value of the change in the average codeword length is $\approx \epsilon$ (equation (6)). This is also seen geometrically in Figure 2. In the second part of the proof, we use that fact to establish bounds on $L_f$.

Codeword lengths are determined as

$$L_k = \left\lfloor \log_2 \frac{1}{P_k} + \theta \right\rfloor \quad (1).$$

This is iterated $n$ times until

$$S = \sum_{k=1}^{M} 2^{-L_k} \leq 1 \quad (2).$$

Since for convergence $\theta \leq 1$, we have $n \leq \left\lceil \frac{1}{\epsilon} \right\rceil$. The average codeword length $L_f$ with the algorithm is

$$L_f = \sum_{k=1}^{M} P_k \cdot L_k \quad (3).$$

Between two successive iterations, $i, i+1$ of (1), $L_f$ is monotonically non-decreasing. The increment in the average word length between two successive iterations is

$$\Delta L_f(i) = L_f(i + 1) - L_f(i) = \sum_{k=1}^{M} (P_k L_k^{i+1} - P_k L_k^i) \quad (4)$$

At iteration $i + 1$, $L_k$ can only increase by 1, since $\theta(i + 1) = \theta(i) + \epsilon$ and $\epsilon < 1$. Let $T(i)$ represent the set of indices of $L_k$ whose length is incremented by 1 at iteration $i + 1$. Then (4) becomes

$$\Delta L_f(i) = \sum_{k \in T(i)} P_k \quad (4').$$

The expected change in the average word length is

$$E\left( \sum_{k \in T(i)} P_k \right) = \sum_{k \in T(i)} E(P_k) = \sum_{k \in T(i)} \frac{1}{M} = \frac{|T(i)|}{M} \quad (5)$$
since all symbols in $T(i)$ are on average neither more nor less probable than the others and $|T(i)|$ represents the number of elements in the set $T(i)$. Let $Q_k^i = \log_2 \frac{1}{p_k} + \theta(i)$ represent the non-integer word length of letter $a_k$ at iteration $i$. With the assumption that $N_p$ is large, that is the alphabet has many distinct probabilities, $Q_k^i$ is a piecewise constant nondecreasing function of $k$, with many small jumps at indices where the probability changes, rather than a few big jumps. For estimation purposes $Q_k^i$ can be approximated by a linear function $R_k$ between indices where $Q_k^i$ crosses an integer value. At iteration $i + 1$, $Q_k^{i+1}$ and $R_k$ are shifted up by $\epsilon$, so the expected fraction of points that cross an integer value at $i + 1$ is $\epsilon$, which can be demonstrated as follows and also seen geometrically (Figure 2).

Let $K$ be the index for which $Q_{K-1}^i < J \in Z_+$ and let $Q_K^i \geq J$. Hence the codeword length crosses an integer value at index $K$ and at iteration $i$. Let $K + m_j$ be the first index at which the word length crosses $J + 1$. So, for $m_j$ symbols the word length $L_k$ is $J$ at $i$ since $L_k = \lfloor Q_k \rfloor$. The function $Q_k$ can be interpolated by a linear function $R_k$ which has value $J$ at $K$ and $J + 1$ at $K + m_j$. At iteration $i + 1$ the interpolation line is shifted up by $\epsilon$, since all $Q_{k\epsilon}$s are incremented by $\epsilon$. Let $L_j$ be the number of symbols with index less than $K + m_j$ for which $R_k + \epsilon \geq J + 1$. It is easily seen using similar triangles, that $\frac{L_j}{m_j} \approx \frac{1}{\epsilon}$ for large $N_p$. Hence within the interval $[J, J + 1]$, the fraction of lengths that change length at iteration $i + 1$ is $\approx \epsilon$. Accordingly, $\sum_{j=1}^{R} L_j \approx \epsilon \sum_{j=1}^{R} m_j = \epsilon M \approx |T(i)|$. Such a linear approximation is done separately for all the integers, in many regions, to cover all indexes. Therefore we conclude that:

$$E[\Delta L_f(i)] = \frac{|T(i)|}{M} \approx \epsilon \quad (6)$$

As $L_f(i)$ increases with $i$, $S$ decreases until at iteration $n$, $S(n) \leq 1$. At this point, as shown in Section 3, the corresponding codewords are uniquely decodable.

We now proceed to establish bounds on $L_f$. We start by first showing that if the sum at the $n$th iteration $S(n) = 1$, then our code is necessarily optimal, in that $L_f = L_b$. We prove this by showing that for both cases, Huffman and Algorithm (A), the underlying tree structure is complete and also that the codeword lengths for each of the symbols are the same, albeit arising from different trees. This equivalence is then used with equation (6) to establish bounds on $L_f$.

Let $S(n) = 1$ and let codeword lengths $L_1, L_2, \ldots L_M = L_m$, obtained with (A2) be sorted in a nondecreasing
order. For establishing completeness of the tree structure, we show that there must be an even number of symbols at all levels. For purposes of a proof, we construct a binary tree for determining the codewords after algorithm A is completed. Starting from $L_m$, we group together successively the two symbols with largest length, which we denote as $a$ and $b$. If $a$, $b$ have the same length $l$, as they are grouped together, a new 'internal' symbol is formed and assigned length $l - 1$. Its codeword is the common prefix of $a$ and $b$; then $a$ may have a suffix 0 attached to it and $b$ a suffix 1. The process is repeated using, randomly, any two available symbols of maximum length. If at any point in the process, there occurs an odd number of symbols with length $l$, then we must pair the last one $a$ of length $l$ with another one $b$ of length $l - 1$. The combined symbol $ab$ must have length $l - 2$ and its code constitutes the prefix of both $a$ and $b$. Then $b$ may have the suffix 0 attached to it and $a$ the suffix 10. However, there is then room for another codeword with suffix 11. But this cannot be assigned to any other symbol, internal or external, since their lengths are less than $l$. Since there is now at least one unused codeword, $S < 1$ necessarily. This contradicts the hypothesis that $S(n) = 1$. Hence we conclude that at any point in the process, the number of symbols with maximal length must be even and hence we have a complete tree. For example, let $r_m$ be such an even number at length $L_m$, and let $r_{m-1}$ be the number of external symbols with length $L_m - 1$, which can also be odd. Once all the pairings at level $L_m$ are completed, there are $r'_{m-1} = r_{m-1} + r_m/2$ symbols at level $L_m - 1$, internal or external, and $r'_{m-1}$ must be even, to assure a complete tree.

Now we introduce some notation, needed to complete the proof. Given $p_m$, the smallest symbol probability, there exists a real number $P_0$, function of $\theta$ such that $p_m < P_0 < 2p_m$, that is implicitly utilized in determining the codeword lengths: algorithm A assigns length $L_m$ to symbols with probability $p$ in the range $(P_0/2, P_0]$, length $L_m - 1$ to symbols with $p \in (P_0, 2P_0]$ and so on.

Now we observe the construction of the Huffman tree. The two least probable symbols, internal or external, are paired together at each step. For convenience, we divide symbols in groups, according to their probability interval. We choose another number $P_1$ such that $p_m < P_1 < 2p_m$, and say that a symbol belongs to the last group $G_m$, if it has probability $p \in (P_1/2, P_1]$, to the next group up, $G_{m-1}$ if $p \in (P_1, 2P_1]$, and so on.

If the size of $G_m$ is even, Huffman coding picks the two least probable symbols, $a$, $b$ both in $G_m$, so they must necessarily have $p \in (P_1/2, P_1]$ and the new symbol $ab$ must have $p_{ab} = p_a + p_b \in (P_1, 2P_1]$; hence it
belongs to \( G_{m-1} \). This process continues until all elements of \( G_m \) are consumed. Instead, if the size of \( G_m \) is odd, then the last symbol in it, \( a \), is paired with another symbol \( b \) in \( G_{m-1} \). Although the size of \( G_m \) is, on average, even half of the times, this disparity may occur in group \( G_{m-1} \), or above. Hence the question arises whether there exists a value of \( P_1 \) such that this disparity never occurs.

If we set \( P_1 = P_0 \); group \( G_m \) contains as many elements \( (r_m) \) as the last level \( L_m \) for algorithm A, and similarly group \( G_{m-1} \) contains \( r_{m-1} \) symbols, as many as in \( L_m - 1 \), and so on. In such a case the Huffman method forms \( r_m/2 \) pairs using all elements of \( G_m \), and after that, group \( G_{m-1} \) will also contain \( r_m/2 \) new internal symbols, together with the \( r_{m-1} \) original ones, for a total of \( r_{m-1} + r_m/2 \). Very likely, the random grouping criterion and the Huffman method will form pairs in different ways, but the final number of symbols at level \( L_m - 1 \) and group \( G_{m-1} \) is the same. It is easy to see, inductively, that this equality would hold at all levels, up to the root. Because of this equality, an external symbol \( a \), assigned length \( l \) by algorithm A is consumed by Huffman coding one level before proceeding with another symbol \( c \), assigned length \( l - 1 \) by algorithm A. So, the lengths of \( a \) and \( c \) will differ by 1. In general, an external symbol assigned length \( l - 1 \) is never paired with one that was assigned length \( l \). Hence, when \( S(n) = 1 \), algorithm A with its random construction of binary trees and Huffman coding generating binary trees based on lowest probabilities, yield codeword lengths that are the same, even if the branches are built differently. Hence, with \( P_1 = P_0 \), we preclude the disparity alluded to earlier.

For the situation where \( S(n) < 1 \), the random tree at some level must have an odd number of nodes. For simplicity, we can assume that this happens at the bottom level \( L_m \), containing symbols in the smallest probability range \( (P_0/2, P_0] \). In case the level at which this disparity (e.g. odd number of nodes) occurs is not the lowest, it will become the lowest at a later stage of the tree construction.

The Huffman criterion will continue to pair two symbols in the range \( P_0/2, P_0 \), until only one (denoted as \( a \)) remains. The symbol \( a \) must then be paired with another one, \( b \), in the probability range \( (P_0, 2P_0] \), to which algorithm A assigned level \( L_m - 1 \). The resulting combined (or internal) symbol may also stay in the same probability range \( (P_0, 2P_0] \) as \( b \), or it may not. It is not possible to predict how the tree structure would evolve and some codeword lengths would likely differ between the two methods. Therefore the random scheme is no longer optimal. Also, note that if algorithm (A) gives \( S < 1 \), there might no value of \( P_1 \) satisfying the
previously questioned property of even number of symbols (internal or external) in each probability group, as a Huffman tree is constructed.

The equivalence of a broad class of codes to Huffman’s one holds only if \( S = 1 \) for a given value of \( \theta \). As an example, consider a set of symbols \( a, b, c, d, e, f, g, h \) with the following probabilities:

0.42, 0.21, 0.1, 0.1, 0.05, 0.05, 0.04, 0.03.

By using (A2) with \( \theta = 0.7 \) we obtain the code lengths: 1, 2, 4, 4, 5, 5, 5, 5, which ensure that \( S = 1 \). The values of \( \log_2 \frac{1}{p_i} + \theta \), before the truncation, are, up to the second decimal digit: 1.95, 2.95, 4.02, 4.02, 5.02, 5.02, 5.34, 5.76. In this case \( p_0 = 0.0508 \), so the largest level \( L_m = 5 \) applies to symbols with \( p \in (0.0254, 0.0508) \), the last four. The Huffman coding procedure would pair \( g, h \) together; then \( e, f \); then \( d + (gh) \), or equivalently \( (ef) + (gh) \). The random criterion of Algorithm (A) may start instead with any pairs formed from \( e, f, g, h \); the final average codeword length is unchanged.

Suppose instead that algorithm A increments \( \theta \) in large steps, (e.g., \( \epsilon = 0.2 \)) and at the last iteration tested, \( \theta = 0.6 \) still generates \( S > 1 \). The next value of \( \theta \) is 0.8, and the lengths after truncation would be: 2, 3, 4, 4, 5, 5, 5. We find that \( S = 0.625 < 1 \); the code can be further improved with algorithm B. After applying it repeatedly to reach \( S = 1 \), the final codeword lengths are: 2, 3, 3, 3, 3, 3, 4, 4. Although \( S = 1 \), such a code has average codeword length greater than Huffman’s. This is because the limit was not reached through algorithm A.

The situation where \( S(n) = 1 \) at iteration \( n \) happens rarely, so in general \( S(n) < 1 \), and \( L_f(n) > L_h \). At the previous iteration \( n - 1 \), we still have \( S(n - 1) \geq 1 \). We know that \( L_f(n - 1) < L_h \) since we established that \( S = 1 \) \( \rightarrow L_f = L_h \) and that \( S \) decreases as \( L_f \) increases. Hence, from (6) it can be deduced that, after \( n \) iterations:

\[ L_f(n) \approx L_f(n - 1) + \epsilon < L_h + \epsilon; \text{ and formally: } L_f \in (L_h, L_h + \epsilon). \]

Specifically,

\[ E[L_f] = L_h + \epsilon/2 \quad (7) \]

when \( L_f \), a function of \( p_1, \ldots, p_m \), is slowly varying and hence can be assumed approximately constant in the
interval \((L_h, L_h + \epsilon)\). The greater the number \(N_p\) of different symbol probability values, the smaller \(\epsilon\) can be for which equations (6) and (7) are still valid, so \(L_f\) is closer to \(L_h\), whenever the alphabet size is large.

To summarize, this analysis suggests that there are possibly many codes with average codeword length equal to Huffman’s. At any step of grouping two symbols into one, we do not need to choose the two least probable ones: it suffices to pick any two in the pool of those with maximal codeword length as generated by algorithm A. One computes the codeword lengths using algorithm A, within a given extra cost \(\epsilon\) with respect to the Huffman code, and then finds a set of codewords, by constructing a tree that has more degrees of freedom than Huffman’s. Such flexibility can be exploited in many ways: for instance to determine a tree with minimum variance as discussed in Sayood [5], more successfully than with Huffman trees, that allow few or even no degrees of freedom.

3. DETERMINING CODEWORDS

Given codeword lengths derived from algorithm A, a Huffman-like tree structure could be constructed, to associate each terminal node with a codeword. However, it is faster to adopt an alternative approach, that has a cost of \(O(M)\). In this setting we do not investigate ways to optimize a code (e.g. as in [5]), other than assuring a prefixed average length.

Let \(L_m\) be the the maximal codeword length and let the symbols be sorted in order of monotonically non-decreasing length. The shortest codeword \(C_1\) has length \(L_1\) and refers to the first symbol. The last codeword \(C_M\) has length \(L_M \equiv L_m\). We map all codewords onto integers between 0 and \(2^{L_m} - 1\) constructing a look-up table \(Lu(i)\) for \(i = 1\ldots M\). We map the most probable symbol with length \(L_1\) as \(Lu(1) = 0\) and the next most probable symbol as \(Lu(2) = 2^{L_m-L_1}\). The rest of the look-up table is constructed in a similar way. Mappings of codes with indices \(i, i + 1\) differ by the amount \(G_i = 2^{L_m-L_i}\); or equivalently \(Lu(i+1) = Lu(i) + 2^{L_m-L_i} = Lu(i) + G(i)\). Hence all mappings can be found by successive increments, until a number \(\leq 2^{L_m} - 1\) is reached. If each mapping \(Lu(i)\) relative to symbol \(i\) is divisible by \(G_i\), then \(\frac{Lu(i)}{G_i}\) is an integer, and its binary representation with 0’s added as prefix if necessary to reach length \(L_i\), can be used as a codeword for each symbol \(i\).
The integer condition is true when the sequence of lengths \( L_i \) is non-decreasing; hence, the symbols are sorted at the beginning. To see this, consider \( Lu(i) \) expressed as the sum of its terms:

\[
Lu(i) = 2^{L_{m-L_1}} + 2^{L_{m-L_2}} + \ldots + 2^{L_{m-L_{i-1}}} = G(1) + G(2) + \ldots G(i-1).
\]

Since \( G(i) = 2^{L_{m-L_i}} \) is a common factor in all of the terms in the summation, the total sum is therefore divisible by \( G(i) \).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>length ( l_k )</th>
<th>( G(i) = 2^{S-L_i} )</th>
<th>( Lu(i) )</th>
<th>( Lu(i)/G(i) )</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( b )</td>
<td>2</td>
<td>8</td>
<td>16</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>( c )</td>
<td>4</td>
<td>2</td>
<td>24</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>( d )</td>
<td>4</td>
<td>2</td>
<td>26</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>( e )</td>
<td>5</td>
<td>1</td>
<td>28</td>
<td>28</td>
<td>11100</td>
</tr>
<tr>
<td>( f )</td>
<td>5</td>
<td>1</td>
<td>29</td>
<td>29</td>
<td>11101</td>
</tr>
<tr>
<td>( g )</td>
<td>5</td>
<td>1</td>
<td>30</td>
<td>30</td>
<td>11110</td>
</tr>
<tr>
<td>( h )</td>
<td>5</td>
<td>1</td>
<td>31</td>
<td>31</td>
<td>11111</td>
</tr>
</tbody>
</table>

**Table 1:** A fast way of obtaining codewords from symbol lengths, sorted in non-decreasing order

As an example, we construct here the code set for the eight symbols \( a, \ldots, h \) seen before. A set obtained this way is uniquely decodable. In fact, the value of \( Lu(i) \) at each step represents the number of leaves of a binary tree with depth \( L_m \) covered at step \( i - 1 \). Hence, starting from \( Lu(i) \), codeword \( C_i \) covers \( 2^{L_{m-L_i}} \) new leaves and it is not the prefix to any other codeword. If \( Lu(i) < 2^{L_m} \) which is true if \( S \leq 1 \), then there is room for all the codewords.

Since the codewords can be uniquely constructed once a listing of codeword lengths is known, such information can be stored in a file header. Also, since the codeword lengths are non-decreasing and many are equal, run-length coding can be employed to store them. The decoder uses the same scheme as the encoder to reconstruct the look-up table and generate the symbols.

Instead of listing all symbols in the data set and their bit patterns, the alternative scheme utilizing symbols
and their lengths can be used, and may be considered analogous to Huffman’s canonical tree, which permits a compact representation of the bit length of each codeword. A reduction in the header size by about a third is possible this way.

4. ENCODING TIME COMPLEXITY

The encoding process consists of Algorithm A, ordering of the symbols by length and generating the codewords. As shown here, all these tasks have a complexity of $O(M)$. The costliest part of all is computing step (A2), the set of $M$ codeword lengths, and then the sum in (A3), also involving $M$ terms. Since both steps are iterated $n$ times, the complexity $C(M)$ is proportional to $M \times n$ and since $n$ depends only on $\epsilon$, the complexity is $O(M)$. Algorithm B, applied only when $M$ is small, has complexity greater than $O(M)$.

Consider for example an alphabet where $M = 2^{16}$ and where $\epsilon = 0.1; \; \epsilon \rightarrow n \leq 10$. In this case $E[L_f] \approx L_h + 0.05$. Then, $C(M) \leq 10 \times M = 10 \times 2^{16}$. This bound is quite loose, since typically $S$ becomes smaller than 1 when $\theta \approx 0.5$, as has been observed empirically. Hence, in the example, choosing $\epsilon = 0.1$ implies $n \approx 5$. With this effect, iterations can be started with $\theta > 0$ to further reduce the complexity. $\theta = 0$ is rarely a good choice, unless all probabilities are reciprocals of power of 2. In that case starting with $\theta > 0$ will result in $S$ too small and $L_f$ unnecessarily large. However, if $M$ is large then this event is so rare that it can be neglected and $\theta = \Theta_0 > 0$ can be a convenient initial choice. The optimal $\Theta_0$, as $M \rightarrow \infty$ can be estimated: as both $M, N_p$ increase, $S(\theta(n))$ becomes smoother (smaller jumps) and it becomes easier to spot a value of $\theta$ giving $S$ close enough to 1 to satisfy equation (7). If $N_p$ is too small, $S(\theta(n))$ is not smooth, and the linear approximation considered in Section 3 as well as equation (7) may not be valid. To obtain a larger $N_p$, probabilities of any two symbols, if equal, could be altered by the addition or subtraction of small constants. This option has not been attempted since for good data bases, the problem has not occurred.

Algorithm B involves a computing time proportional to $2^{L_m} \times (1 - S)$, because at each step $S$ is incremented by $2^{-L_m}$, and this can be done while $S < 1$. The first factor grows with data size, and therefore it is $O(N)$, where $N$ is the number of symbols, with repetitions, in the data set. Since the empirical probabilities $p_k$ are derived from the occurrences of symbols in the data, if $a_k$ occurs only once, it has a probability $p_k = \frac{1}{N}$. A symbol is often associated with a tuple of characters, as described in Section 5. For purpose of establishing
a lower bound to empirical probabilities, instead of encoding any symbol occurring at least once in the data set, we consider only tuples occurring at least $n$ times; otherwise they are decomposed into smaller parts, encoded as distinct symbols. We found $n = 5$ convenient in a practical implementation. Then the smallest empirical probability is $\frac{1}{N}$, unless the symbol in question is not decomposable. Then it can be assigned a smaller probability. In any case, it is possible that for large databases $2^{L_{m}} = O(N/n) > M$. Next, for a fixed $\epsilon$, the second term $(1 - S)$ typically decreases with $M$, as $S$ gets closer to 1 at the end of algorithm A, until the relation (7) can be assumed valid. Then the term $(1 - S)$ stabilizes to a value less than $\epsilon \log 2$, as can be derived analytically. So the total time for algorithm B can become greater than $O(M)$, as $N \gg M$. However, it has been observed that for large data sets, adequate convergence is obtained with algorithm A and algorithm B is often not needed.

For ordering symbols by length, a procedure that employs distribution sort can be achieved in $O(M)$ steps, as shown by Cormen [6]. Note for this purpose that while in Huffman coding symbols are ordered by probability, this is not necessary for the Length-first-coding scheme. Once $L_{1}, \ldots, L_{m}$ are found with algorithm A, the number of possible values is only $L_{m} - L_{1} = O(\log(N/n))$, which is often several orders of magnitude less than $M$ and $N_{p}$. In this setting, distribution sort is the fastest tool for ordering.

The cost of formulating the codewords using a look-up table and mapping operation is also $O(M)$. The table values $L_{m}(i)$ are found with $M$ successive increments, and the codewords are then derived using $M$ shift operations, which are faster than divisions.

5. DEFINING THE DICTIONARY

Before encoding the data, symbols that constitute the alphabet for the given data set must be determined. The dictionary consists of the alphabet, which needs to be transmitted to the decoder; so it is typically stored in the header file, or, when several files are to be coded, in a separate file. In order to exploit inter-character redundancy in the data, it is necessary to consider an extended alphabet, consisting of sequences of more than one character. Sequences of $N$ characters however, cause the alphabet to grow as $2^{N}$ which can seriously affect the computation time. To allow good compression, we need both a good model of the data and an efficient way of representing (or learning) the model. Starting from the simplest concept of
associating each ASCII character with a symbol, we have developed several alternatives where the more complex schemes lead to better compression but not necessarily at the expense of a relatively larger header file. Methods to define symbols in the alphabet are as follows:

1. All ASCII characters are defined as symbols in the alphabet.

2. In order to exploit the correlation between adjacent letters, we allow as symbols, sequences of two consecutive bytes. The statistical structure is then the set of joint probabilities $p(i, j)$, the probability of event $i, j$ with indices $i, j$ ranging over all possible symbols. This of course increases the number of symbol indices that need to be written in the header - that of all 24-tuples of 2 bytes in the data, along with single ASCII characters.

3. A refinement to the scheme above consists in restricting the 2-tuples that are assigned code numbers, only to “alphabetical” pairs, i.e. where both characters are either letters in the alphabet, digits or punctuation. Formally, they are both in a subset of $N_2$ selected characters, and $N_2$ is chosen after the maximum index $M$ is determined. This restriction allows a reduction of the number of symbols from $M = 256 + 2^{16}$ to $256 + N_2^2$. If $r$ bits are assigned to the header for addressing the presence of any tuple in the data set, then $M = 256 + N_2^2 \leq 2^r$. The set $N_2$, determined by the user, contains letters most likely to occur in ordinary files; so the probability of finding a pair not present in the set of $N_2^2$-tuples is low. In this setting, all the 256 ASCII characters and the selected pairs are valid tuples, and if present in the data, are assigned an index.

4. The next order of complexity involves letter sequences of length three. A set of joint probabilities $p(i, j, k)$ are required for 3-tuples. As before, the letters in any triple can be restricted in that they belong to a more constrained set of $N_3 < N_2$ symbols. Accordingly, the size of the alphabet becomes:

$$M = 256 + N_2^2 + N_3^3$$

The $N_3$ bytes can be chosen as the set of small-case letters, the space, newline, period (.), comma (,), apostrophe (') and maybe a few others.

5. Next, an adaptive algorithm is formulated, where the characters that are used to form the $n$-tuples are selected based on their frequency of occurrence in the data set. In the implementation of this algorithm,
all bytes in the file of the data set are first counted. Then a function \(Sel\) arranges in order of decreasing occurrence, the 256 ASCII characters in a 256–dimensional vector. An inverse function \(Map\) maps the ASCII characters to their order in the vector. For example, if ‘space’ (ASCII 32) and ‘e’ (ASCII 101) have, in order, the greatest frequency of use in a data set, then we have:

\[Sel(0) = 32; Sel(1) = 101; \text{ and } Map(32) = 0; Map(101) = 1.\]

Then, the first \(N_2\) characters in the array \(Sel\) are used to form 2-tuples and the first \(N_3\) characters used to form 3-tuples. The process can be continued to form 4-tuples where each element of the 4-tuple is among the \(N_4\) most used bytes and so on. However, doing so can increase the number of code words to a point where both the computation complexity and compression are adversely affected.

(6) An alternative scheme to the direct implementation of selection of the \(n\)-tuples in the alphabet, is a double mapping scheme where the 3-tuples and 4-tuples symbols are constructed as follows:

In the file, we count the occurrences of those pairs belonging to any of the \(N_2^2\)-tuples, and arrange them in decreasing order. The first \(G_3 < N_2^2\) of them are then selected. Each 3-tuple then starts with a pair in \(G_3\) with the third letter from among the first \(N_3\) occurring bytes, as before. For constructing 4-tuples, we select \(G_4 < G_3\) of such pairs; 4-tuples are formed by concatenating any two pairs in \(G_4\). The advantage of a double mapping scheme over the single one, is that it allows many more sequences to be constructed as symbols while keeping the maximum index low.

Setting all the constants in the double mapping scheme so that we restrict \(M \leq 2^{16}\), allows the symbol indices to fit into 2 bytes. The disadvantage of the scheme in (6) might be its increased complexity and consequently, a longer processing time. Moreover, for the schemes (5) and (6) the header must also include all single character selections and for the scheme (6) the pair selections as well. However, this extra overhead is much smaller than the original header.

Algorithm (6) can be extended to triple mappings, to encode sequences of 5 letters, as combinations of selected 3-tuples and 2-tuples and to sequences of 6 letters, as combinations of 2 selected 3-tuples, etc. Any of these extra features could improve compression a bit more, but at the expense of greater complexity. They
could be applied to encode very large data sets, in which case $M$ would be set to a larger value (like $2^{20}$). We have only considered approximations up to the fourth order.

6. DISCUSSION: APPLICATION TO BULK COMPRESSION & EDITING COMPRESSED FILES

6.1. Bulk Compression

Modeling the data with higher-order statistics allows exploitation of correlation within the data. The corresponding increase in the number of symbols results in an increase in the header size of the compressed file, since each symbol index is specified by 2 bytes. Assuming that the order of the statistics in modeling the given data set is fixed, the header file grows at a nonlinear rate with the file data size: adding more data to a large file generally involves few new symbols and hence only an incremental increase in the header size. Accordingly, the percentage of space occupied by the header in a compressed file decreases with the data size. The high-order algorithms as in (6) are suitable for large data files, while those in (3) which use pairs but not triples, etc. involve smaller headers and therefore are appropriate for small files. However, the simpler algorithms cannot fully realize compression gains from the redundancy inherent in the data.

In order to realize the gains of the higher order algorithms as in (6), but to mitigate the effects of the larger header size when working with smaller files, the algorithm was applied to a bulk compression application, defined as the compression of very large numbers of small homogeneous files. The algorithm scans all files in a directory, counts the symbols and stores the indices and code lengths in a unique header. Each file is then compressed using the same codewords. The Decoder accesses this header to reconstruct the original files. If all of them have a fairly homogeneous text, and are of small to medium size, the $Lfc$ algorithm is seen to outperform many other UNIX based compression programs in the examples tested. Results are shown in Section 6.3.

Files with similar statistical structure are defined as being homogeneous. Hence, natural written languages, like English and French should be scanned separately and the relative symbol indices stored separately in different headers. It may help to put together files with the same extension and use escape characters, like
"*" or ":?" in the arguments; for example, source files listed as "*.c" or documents listed as "*.tex" could be scanned together.

6.2. Editing Compressed files

As another application, it may be desired to edit files already compressed. Symbols not previously indexed in the header may be introduced, that correspond to new pairs, within the $N_2^2$ ones that could have a valid index assigned to them. Alternatively, they could be triples, within the $N_3^3$ selected ones, etc. These symbols cannot be used, because they do not have a codeword assigned to them. However, the edited files could still be compressed with the existing header, since all ASCII characters are always present as valid symbols, and the new pairs, triples, etc may be coded as separate characters.

The compression would not be as efficient as decoding all files, reassigning all codewords and compressing again; but doing so increases the complexity substantially. There are two alternative procedures for re-encoding an edited file; without having to re-scan the whole data base, while still maintaining a fairly good compression ratio.

(a) When computing the sum $S$ in A3 for the whole data base of files, set a limit for $S$ less than 1, such as $1 - 2^{-3} = 0.875$. Then, as the edited file is scanned and new symbols generated, there is place for them. A local, very small header will indicate only the new indices and the new code lengths. All new codes will start with the prefix ‘111’ in case the previous limit was $S \leq 0.875$.

(b) Keep the limit $S \leq 1$. If a symbol generated in the new file is not in the data base and it is relative to a pair, then encode the 2 bytes separately. Each byte must have a code. If the symbol generated is relative to a triple, then encode the first two letters as defined in the header and the third can start a new sequence. Similarly, a quadruple can be broken into 2 pairs, or form a triple and the 4th letter may start a new sequence. For instance, suppose there is the new word pattern “blue sky” in a newly edited file and the previous sequence starts at ‘b’. If “bl" and “ue" are both selected pairs, then the algorithm has a valid index for “blue”, but it is not in the header. Then the encoder looks for the index to “blue". If it exists, then encodes it, then, tries the triple “es”, that is valid since the pair composed of ‘e’ and ‘s’ is probably
selected (if the quadruple “e sk” is valid, also that is tried). If also “blu” is not indexed, then encode just “bl” as distinct symbol, then continue with “ue...”. This method is more practical and does not involve extra headers.

6.3. Benchmarks

Two files *book1* and *book2* in normal English in fiction and non-fiction, obtained from the Calgary/Canterbury text compression corpus were used for compression. The Tables indicate the compressed file size in bits. CPU times refer to both user and system. Results are given for both single file compression and bulk compression using *Lfc* and standard compressors. Table 2 gives the results of compressing *book1* as a single file.

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>Lfc</th>
<th>gzip</th>
<th>compress</th>
<th>yabba</th>
<th>whap</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>book1</em></td>
<td>768771</td>
<td>361175</td>
<td>313376</td>
<td>332056</td>
<td>320662</td>
<td>338046</td>
</tr>
<tr>
<td></td>
<td>CPU</td>
<td>12.4</td>
<td>12.22</td>
<td>5.28</td>
<td>5.04</td>
<td>3.99</td>
</tr>
</tbody>
</table>

Table 2: Compression of *book1* as a single file

In Table 3, *book1* is divided into 83 chapters of 200 lines each and all 83 files are compressed.

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>Lfc</th>
<th>gzip</th>
<th>compress</th>
<th>yabba</th>
<th>whap</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>book1</em></td>
<td>768771</td>
<td>362037</td>
<td>371398</td>
<td>422613</td>
<td>401148</td>
<td>419758</td>
</tr>
<tr>
<td></td>
<td>CPU</td>
<td>14.7</td>
<td>9</td>
<td>5.04</td>
<td>19.47</td>
<td>16.24</td>
</tr>
</tbody>
</table>

Table 3: Bulk compression with 83 files of *book1*

In Table 4, *book1* is divided into 165 chapters of 100 lines each and 1 chapter with 122 lines. Here *Lfc* performs about 7 % better than *gzip* and its overall size has increased less than 600 bytes compared to that of *book1* coded as a single file.

In Table 5 shows compression results with size only, using *book2* consisting of 157 chapters of 100 lines each. Here, bulk compression does not perform as well as *gzip* even when using small files. That could be due to the significance of higher order redundancies in *book2*. Entropy associated with sequences longer than 4 letters may be significant but it is not recognized by *Lfc*.
<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>Lfc</th>
<th>gzip</th>
<th>compress</th>
</tr>
</thead>
<tbody>
<tr>
<td>book1</td>
<td>768771</td>
<td>362357</td>
<td>390872</td>
<td>451201</td>
</tr>
<tr>
<td>cpu</td>
<td>16.2</td>
<td>11.2</td>
<td>5.84</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Bulk compression with 166 files of book1

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>Lfc</th>
<th>gzip</th>
<th>compress</th>
</tr>
</thead>
<tbody>
<tr>
<td>book2</td>
<td>610856</td>
<td>302558</td>
<td>280073</td>
<td>355021</td>
</tr>
</tbody>
</table>

Table 5: Bulk compression with 157 files of book2

Finally we give an example of the versatility of Lfc, by applying it to new, but homogeneous files, that have not been used in establishing the alphabet. This has application in databases where either new files are added or existing files are edited. In the example, book1 consisting of chapters 1-79 was compressed with Lfc and its alphabet stored in the header. New files consisting of chapters 80-83 were encoded using the same header. In Table 6 we see 11% to 14% less memory used than that with gzip.

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>Lfc</th>
<th>gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>chapter80</td>
<td>9283</td>
<td>3064</td>
<td>4558</td>
</tr>
<tr>
<td>chapter81</td>
<td>8904</td>
<td>3844</td>
<td>4369</td>
</tr>
<tr>
<td>chapter82</td>
<td>9300</td>
<td>3935</td>
<td>4456</td>
</tr>
<tr>
<td>chapter83</td>
<td>9448</td>
<td>4298</td>
<td>4759</td>
</tr>
</tbody>
</table>

Table 6: Bulk compression with new files

7. CONCLUSION

This work has presented a new and simple prefix property type iterative algorithm. We have shown that the average code word length is typically slightly longer than that obtained with Huffman coding, but can be made arbitrary close to Huffman. As $M \rightarrow \infty$, the encoding algorithm complexity is shown to be of $O(M \times n)$ where $n$ is independent of $M$. 
The algorithm has been found to be well suited to application in bulk compression, where it is desired to compress a large number of small homogeneous files. It is seen that compressing jointly many files of size less than 10 K-bytes each using Lfc, often causes a saving of about 10% of memory when compared to compressing all of them separately using compress. The advantage becomes greater as the file sizes become smaller, since compress and other UNIX based compressors are more effective for large files.

Another application shows a particular advantage of Lfc with its property of employing global data bases. New or edited files can be compressed using an alphabet derived earlier from previous files and stored in a global data base. In the example shown, we see gains of 11% to 14% over that using gzip.

8. REFERENCES


Figure 1. Algorithm A and B.

Figure 2. Piecewise constant function $Q_{k}^{i}$ and linear function $R_{k}^{i}$. 