Adaptive Minimum-Redundancy Coding over Binary Channel with Unequal Cost Letters : Implementation and Analysis

Chi-Huan Tsai

July 23, 2004

Abstract

In this paper we consider the adaptive case of prefix-free coding with unequal letter costs. That is, given an alphabet with size n*, and a binary channel with unequal cost digits, we try to construct a prefix-free code* W *adaptively. Our goal is to achieve or approximate minimum-redundancy a tree-based method can generate.*

Our experimental results show that generally our adaptive algorithms can construct codes with low redundancy.

.

Key Words: information theory, source coding, unequal cost, adaptive algorithm, binary channel.

Contents

List of Figures

Chapter 1

Introduction

In this paper we consider the adaptive case of prefix-free coding with unequal letter costs. That is, given an alphabet with size n, and a binary channel with unequal cost digits, we try to construct a prefix-free code W adaptively. Our goal is to approximate the minimum redundancy codes.

The original Karp's problem[13] is an optimization problem and many research papers have tried to solve it[1]. But the general case solution still needs exponential time complexity so far, thus not suitable for practical application. Basically, we study the approximation method of the adaptive case of Karp's problem for two reasons:

1. Nowadays, the adaptive data compression(e.g. adaptive Huffman coding, adaptive arithmetic coding) is proved to be more useful than static compression methods for some specific applications.

2. The adaptive case is essentially approximate.

Since 1948 Shannon's fundamental theorem proved that there do exist codes which achieve sharp upper bounds on the transmission rates over discrete noiseless channel, researchers had tried to construct the codes. Huffman[12] first proposed a tree-shaped structure that can construct the optimal code. That is, given an alphabet with n source symbols $\{\alpha_1, \ldots, \alpha_n\}$, each symbol α_i with probability p_i, we can use Huffman tree to build the codeword mapping W that minimizes the total length of stream transmitted over the discrete noiseless channel.

However, Huffman's minimum-redundancy coding only applies to channels with equal letter length. Sometimes, letters may have different features other than "length". For example, over a binary channel, to transmit digit 1 may cost more energy than digit 0, thus digit 1 and digit 0 have different *'costs'*, in the extreme case, the cost of a digit could be 0[8].

Later in 1961, Karp[13] formally stated the unequal letter costs problem. Given an alphabet set A with n source symbols $\{\alpha_1, \ldots, \alpha_n\}$, without loss of generality, let the source symbols have probabilities $p_1 \le \ldots \le p_n$, and the costs of the r letters $\{l_1, \ldots, l_r\}$ have $c_0 \le \ldots \le c_{r-1}$. A codeword w_i for α_i is a string with letters $l_i's$. We define $Num_j(w_i)$ as the number of letter l_j which appears in w_i. Naturally, the cost of a codeword w_i is $cost(w_i) = \sum_j c_j \cdot num_j(w_i)$. A code W is a collection of codewords. $COST(W) = \sum_i p_i \cdot cost(w_i), w_i \in W$.

Definition 1.0.1. A code W is *optimal* if $COST(W)$ is minimum for all possible code.

Definition 1.0.2. A code is *prefix-free* if no codeword is a prefix of any other codeword. We call such a code *prefix code.*

Note that we can construct a n-leaf full tree T (called a *parsing tree*), assign a symbol α_i to certain leaf node, and the path from root to the leaf node defines the codeword w_i for α_i. Every parsing tree T represents certain prefix code W, but not all the prefix code can be represented by a parsing tree. In this paper we focus on the prefix codes which can be represented by a parsing tree.

Karp's problem is: *can we efficiently construct an optimal prefix code?* Karp thought of it as an integer programming problem, but the algorithm needed exponential time. Since then, there is still no known polynomial time algorithm for Karp's problem. We don't even know whether it's an NP-Hard problem in essence. The best known solution is provided by Golin and Rote[11], the algorithm uses dynamic programming technique and gives the solution in $O(n^{C+2})$, where n is the size of the input alphabet and C is the largest letter cost. This polynomial time algorithm, however, can only deal with special case that c_i's are all integers. Moreover, when C is very large the running time is unacceptable. Bradford et. al.[7] later improved the algorithm, but considered only binary channel with costs c_0 and c_1. The time complexity was reduced to $O(n^C)$. When c_0 is 1 and c_1 is 2, time complexity is improved from $O(n^4)$ to $O(n^2)$[1].

Golin et al.[10] have also developed a scheme that approximates Karp's problem in polynomial time. This approximation algorithm has time complexity $nd\log(n)exp(O(\frac{\ln(1/\varepsilon)^2}{\varepsilon^2}))$ to find a prefix code of cost at most $(1+\varepsilon)\mathsf{OPT}$, where OPT stands for the optimal cost.

[1]This is the simplest case these algorithms can solve and the most apparent improvement is shown.

Besides the above algorithms, heuristic functions for constructing the parsing tree are also considered in [4, 9, 14, 15]. These heuristic functions don't give exact solution, but try to approximate it to some extent. Two heuristic functions will be analyzed and implemented in this paper. Mehlhorn[15] proposed a top-down approach to build the parsing tree for approximating the optimal cost. While Gilbert[9] proposed a bottom-up approach. We'll describe their original algorithms in chapter 2, and provide our improvement on these methods in chapter 3.

The above all relates to static model. We will also discuss the performance of the above algorithms when applied to adaptive environment. We introduce two adaptive methods, based on the top-down approach and bottom-up approach, respectively. Our algorithms can encode and decode data *adaptively over binary channel with unequal cost letters*. The experimental results will be shown in chapter 4. Bradford et al.[7]'s algorithm will also be implemented in order to acquire the optimal solution for comparison with our approximation algorithms.

Chapter 2

Preliminaries

The capacity C of a discrete channel is defined as [18]:

$$C = \lim_{T\to\infty} \frac{\log N(T)}{T}, \tag{2.1}$$

where T is the duration of the channel and $N(T)$ is the number of all possible letter streams the channel can serially transmit in duration T, that is, the maximum number of distinguishable letters. Here we can simply think of duration as some kind of "cost", and in our case the costs could be different. Suppose there are n letters with durations $\{t_1, ..., t_n\}$. As we know the last letter of an output stream can be any one of the allowed letter on the channel, thus

$$N(T) = N(T - t_1) + \ldots + N(T - t_n) \tag{2.2}$$

Let X_0 be the largest real solution of the following equation:

$$X^{-t_1} + \ldots + X^{-t_n} = 1 \tag{2.3}$$

When t is large, $N(t)$ will be near X_0^t, and plugging it into equation 2.1, we will have $C = \log X_0$. Note that the entropy of a source distribution is defined to be:

$$H = -\sum_i p_i \log p_i \tag{2.4}$$

Entropy is the expected number of letters needed to transmit a symbol. Thus, in our noiseless binary channel, since in every single time(i.e. cost) unit at most $\log X_0$ letters can be transmitted, the maximum amount of symbols we can communicate through it is $\frac{\log X_0}{H}$. That is, we can transmit a symbol with cost at least $\frac{H}{\log X_0}$.

Therefore, after we can generate a code W for the source distribution, we can measure its efficiency as below. First, compute the entropy H according to source distribution, and X_0 according to channel characteristic. Second, estimate the cost of the transmission while we apply the code to the source distribution: $COST(W) = \sum_i p_i \cdot cost(w_i), w_i \in W$.
Compare the performance of the code $COST(W)$ with the expected rate $\frac{H}{\log X_0}$ to judge the efficiency of the code W. The closer they are, the less redundancy is induced.

We will describe three algorithms proposed before dealing with the unequal letter cost problem. Note that the code they generate are all tree-based(i.e. can be represented by a tree). The main drawback of tree-based coding technique is that we can only map a symbol to a codeword of letters with integral length. As for other compression algorithms(e.g. arithmetic coding), the limitation doesn't exist and thus can be asymptotically close to entropy. However, by now the only known algorithms for solving unequal letter costs problem are all tree-based. It's an open problem that if there exists a good algorithm which is "not" tree-based.

2.1 Digital Expansion Method

Consider the Shannon-Fano encoding method first. For a source distribution $p_1 \leq \ldots \leq p_n$, the algorithm splits the probabilities into two piles, each has sum of probabilities as close to $\frac{1}{2}$ as possible. Then assign letter '0' to one pile and '1' to the other. Continue to split probabilities and assign letters until a pile of only one probability remains.

For example, consider a probability distribution $\{0.1, 0.2, 0.3, 0.4\}$, we can encode as follows:

0.1	0	0	0
0.2	0	0	1
0.3	0	1	
0.4	1		

The expected number of a codeword is 1.9 bits.

Note that if we try to take 0.1 and 0.4 together at the first step, in order to approach $\frac{1}{2}$, the resulting code will be:

0.1	0	0
0.4	0	1
0.2	1	0
0.3	1	1

The expected bit number of a codeword is 2. Also note that if in every step we can split equal size piles, the parsing tree will be exactly the Huffman tree and is the optimal one.

Now we consider the unequal letter cost case. Suppose we have only two symbols to be transmitted over a binary channel, and their probabilities are p and $1-p$, respectively. What p should be if we want to transmit with the highest rate? In the beginning of this section, we have shown that $COST(W) \geq \dfrac{H}{\log X_0}$(remember X_0 is the largest real root of equation 2.3) and our goal is to find the code with least $COST(W)$. If the source distribution is variable, we can estimate how good the splitting method can be. We know $\dfrac{H}{COST(W)}$ has an upper bound $\log X_0$, but when achieved?

Just replacing p and $1-p$ with $X_0^{-c_0}$ and $X_0^{-c_1}$, respectively, we can see that

$$\begin{aligned} \frac{H}{COST(W)} &= \frac{-p\log p - (1-p)\log(1-p)}{c_0 p + c_1(1-p)} \\ &= \frac{-X_0{}^{-c_0}\log X_0{}^{-c_0} - X_0{}^{-c_1}\log X_0{}^{-c_1}}{c_0 X_0{}^{-c_0} + c_1 X_0{}^{-c_1}} = \log X_0 \qquad (2.5) \end{aligned}$$

Thus when $p_0 = X_0^{-c_0}$ and $p_1 = X_0^{-c_1}$, we'll have the highest information transmission rate.

We try to split the probabilities as follows. For the probabilities $p_1 \leq \ldots \leq p_n$, we split them as $p_1 \leq \ldots \leq p_i$ and $p_{i+1} \leq \ldots \leq p_n$, where $\sum_1^i p_k$ is as close to $X_0^{-c_0}$ as possible. Then we separately deal with the two piles recursively, expand forward the deeper level. Expanding stops while there is only one probability in a pile, and we can assign a codeword to the symbol with that probability. In the same spirit, we expect that splitting like this at every stage would generate good parsing tree(i.e. nearly optimal prefix code).

2.2 Generalized Huffman Method

In the beginning of this chapter, we have seen that the best code W is the one with *cost* equivalent to the lower bound. We call it the *zero-redundancy* coding.

Apparently not all source can be coded without redundancy. However, a source which can be coded without redundancy has its own interesting characteristics. Since in this paper we focus on the tree-based coding methods, naturally we would ask: what characteristics does a parsing tree have if it's regarding to a zero-redundancy coding?

In the last section, in each stage of digital expansion method, the probabilities are separated into two piles, and we have seen that when these two piles of probabilities have ratio $(X_0^{-c_0} : X_0^{-c_1})$, the optimum is achieved, thus zero-redundancy. Therefore, digital expansion method assumes source can be coded without redundancy and split the piles as if the ratios are always ideal $(X_0^{-c_0} : X_0^{-c_1})$.

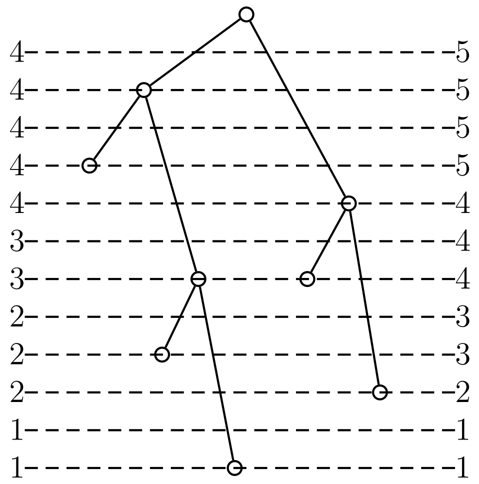
Digital expansion method builds a parsing tree with a top-down approach. Actually, we may build a parsing tree with a bottom-up approach too. Also the zero-redundancy characteristic will be used to be the assumption. In our view, essentially generalized Huffman method and digital expansion method are similar, in spite of the difference in building the parsing tree.

In [2], zero-redundancy coding occurs for those distributions $p_1, \ldots, p_n$ for which:

$$\begin{aligned}
& \sum_{i=1}^{n} p_i \cdot cost(w_i) = \frac{H}{\log X_0} = \frac{-\sum\limits_{i=1}^{n} p_i \log p_i}{\log X_0} \\
\Rightarrow\ & \log_{X_0} p_i = -cost(w_i) \\
\Rightarrow\ & p_i = X_0{}^{-cost(w_i)} \\
& = X_0{}^{-\sum\limits_{j=0}^{r-1} c_j \cdot num_j(w_i)} \\
& = X_0{}^{-c_0 \cdot num_0(w_i) - c_1 \cdot num_1(w_i)} (\because \mathit{binary\ channel}) \\
& = (X_0^{-c_0})^{num_0(w_i)} \cdot (X_0{}^{-c_1})^{num_1(w_i)} \qquad (2.6)
\end{aligned}$$

Any pair of siblings have the same path from root to their parent, and different in the edges between parent and the leaves. Thus for a zero-redundancy code over binary channel, any pair of siblings in its parsing tree must have ratio $X_0^{c_1-c_0} > 1 (\because\ X_0 \in (1, 2])$.

The algorithm mentioned in [9] tries to approximate the zero-redundancy code with the above idea. The only difference between Huffman coding and



left side: B sequence(from bottom to top)
right side: each $N_k(B)$ on level k

Figure 2.1: $c_0 = 2, c_1 = 5$

generalize Huffman method is the way they merge the nodes. In Huffman coding, the two symbols with the least probability are merged. In generalized Huffman method, one symbol has the least probability p_0, and the other has probability as close to $p_0 \cdot X_0^{c_1-c_0}$ as possible.

2.3 Dynamic Programming Approach

In [7], dynamic programming technique is used to solve the unequal costs problem, where the costs can only be integers.

Suppose a tree T has left link with length(cost) c_0, right link with length(cost) c_1, where c_0 and c_1 must be integers, and $c_1 > c_0$. Then at every depth, certain leaf nodes or internal nodes could locate there. Let B be the sequence $(b_0, ..., b_{d-1})$, where b_i is the number of right children and d is the depth of T, no matter leaf nodes or internal nodes, at or below level i. Note that level 0 is the first level with right children and thus the lowest. Say B is the characteristic sequence of T. B is always monotonic(i.e. $b_0 \leq \ldots \leq b_{d-1}$). We show an example in Figure 2.1.

Let $N_k(B)$ be the number of leaf nodes at or below level k. We know every leaf nodes in the parsing tree T may represent some symbol, and to achieve lower cost, symbol with lower probability shall be located at lower level. Thus $N_k(B)$ is the number of symbols with the lowest probabilities which will be allocated at or below level k. And $N_k(B)$ is decided by sequence B as follows:

Theorem 2.3.1. $N_k(B) = b_k + b_{k-(c_1-c_0)} - b_{k-c_1}$

Proof. At first we add the right children at or below level k. But left children must also be considered. Apparently for any right children at or below level $k - (c_1 - c_0)$, it must have a sibling as the left children at or below level k. However the internal nodes could be counted in and we need to subtract them. Similarly, for any right children at or below level $k - c_1$, its parent must be at or below level k. The proof is complete. $\square$

Next we define $cost(B, P)$ as the cost if we assign symbols with probability distribution P to the parsing tree T that has characteristic sequence B. The depth of a leaf node is equivalent to the cost of the node's regarding codeword. In Figure 2.1, the tree has 5 leaves with depth $\{12, 10, 9, 7, 4\}$ respectively. If the 5 symbols have occurred for $\{2, 3, 4, 5, 10\}$ times respectively, after assigning them to the tree, we have $cost(B, P) = 6.875$. We can decide $cost(B, P)$ by $N_k(B)$.

Theorem 2.3.2. $cost(B, P) = \sum_{0 \le k < d} S_{N_k(B)}$, *where* $S_i = \sum_{j \le i} p_j$.

Proof. To calculate the $cost(B, P)$ when we assign symbols that has probability distribution P to the parsing tree T that has characteristic sequence, first we should know the depth of leaf nodes and their corresponding symbols. Then $cost(B, P) = \sum_i depth(i) \cdot prob(i)$. $\square$

Suppose that a symbol a_i with probability p_i is located at level k'. To calculate $cost(B, P)$, at every level k, if p_i is among the smallest $N_k(B)$ probabilities(i.e. $k' \le k$), p_i must be added once. After every level has been through, p_i is added for $(d - k')$ times, that's the depth of the symbol, and thus we can calculate $cost(B, P)$ by this method.

When parsing tree T is available, we can make its characteristic sequence B easily. But what if we have only the sequence B? Can we establish T by B?

In fact, the inverse direction is easy too. First process b_0 of sequence B. Indexing the nodes by their cost, let k be the index of the lowest left leaf node. k must be $b_{c_1-c_0-1} + 1$ since there are no left children below level

$c_1 - c_0$. Thus, p_1 can be assigned to one of the b_0 leaf nodes, and p_k can be assigned as the left sibling. Their parent internal node then becomes a leaf node. Change the B sequence. Add probability $p_1 + p_k$ to the set of probabilities. Then process next element of sequence B in the same way, until all the probabilities are assigned properly.

Therefore all we need to do is to establish characteristic sequence B with minimum cost. Then build the parsing tree T by processing B.

For a c_1-elements section of sequence B, let it be a c_1 tuple $(i_0, \ldots, i_{c_1-1})$, $0 \leq i_0 \leq \ldots \leq i_{c_1-1} \leq n-1$. Note that each element is smaller than $n-1$ since there are only n symbols in the alphabet, there is no need to consider a parsing tree with more than n leaf nodes, there are at most $n-1$ internal nodes and thus at most $n-1$ right children.

When we move the section(or window) from bottom to top, the new c_1 tuple we get is $(i_1, \ldots, i_{c_1-1}, i_{c_1})$, where i_{c_1} must be at least i_{c_1-1}. And this is just like the way we described in Theorem 2.3.2. Therefore we have:

Theorem 2.3.3. *By every move-up, the cost shall increase* $S_{i_{c_1}+i_{c_0}-i_0}$.

Proof. As in Theorem 2.3.2, by every move-up the cost increases $S_{N_k(B)}$. According to Theorem 2.3.1, $N_k(B) = b_k + b_{k-(c_1-c_0)} - b_{k-c_1} = i_{c_1} + i_{c_0} - i_0$, ($k = c_1$, since level c_1 is just climbed onto). Our statement is proved. $\square$

When all the elements in the c_1 tuple are $n-1$, we stop. This process shall compute the $cost(B, P)$.

We restate the data structure to be a graph and then solve it as a shortest path problem. Let any c_1 tuple be a vertex. And if first $c_1 - 1$ elements of vertex u and last $c_1 - 1$ elements of vertex v overlapped, there is an edge between u and v, and its weight is $S_{i_{c_1}+i_{c_0}-i_0}$. It's easy to see the shortest distance between vertex $(0, \ldots, 0)$ and $(n-1, \ldots, n-1)$ is $cost(B, P)$.

For example, let $c_0 = 2, c_1 = 3, n = 5$. To find the shortest path we may go through:

$$\begin{aligned}(0,0,0) &\overset{S_1}{\to} (0,0,1) \overset{S_3}{\to} (0,1,2) \overset{S_4}{\to} (1,2,2)\\ &\overset{S_4}{\to} (2,2,3) \overset{S_4}{\to} (2,3,3) \overset{S_4}{\to} (3,3,3)\\ &\overset{S_4}{\to} (3,3,4) \overset{S_5}{\to} (3,4,4) \overset{S_5}{\to} (4,4,4)\end{aligned}$$

This produce the sequence $(1,2,2,3,3,3,4,4,4)$, $cost(B,P) = cost(4,4,4) = S_1 + S_3 + 5S_4 + 2S_5$.

The graph has $O(n^{c_1+1})$ edges, thus need $O(n^{c_1+1})$ time. Time complexity can be further improved to $O(n^{c_1})$ and it needs much more complicated processing.

Let $\delta = (i_1, \ldots, i_{c_1-1})$ be a $(c_1 - 1)$-tuple. Define a matrix A_δ as follows:

Definition 2.3.4. $A_\delta(i,j) = \mathit{cost}(i, i_1, \ldots, i_{c_1-1}) + S_{j+i_{c_0}-i}$

A_δ can be used to compute cost of c_1-tuples.

Theorem 2.3.5. $\mathit{cost}(\delta, j) = \min_i\{A_\delta(i,j)\}$

Proof. δ is the overlapped part of tuple (δ, j) and (i, δ), that is, from (i,δ) to (δ, j) we are moving one level up. And the only possible previous section of (δ, j) is (i, δ), where $j \geq i_{c_1}$ and $i \leq i_1$. Thus the $\mathit{cost}(\delta, j)$ can be calculated by one of them. By Theorem 2.3.3, we know that $S_{j+i_\alpha-i}$ is the cost needed for moving up one level. Therefore we can calculate $\mathit{cost}(\delta, j)$ by means of processing matrix A_δ, that is, when column j is given, find the row i which has the minimum. □

Note that A_δ is a *Monge* matrix:

$$A_\delta(i,j) + A_\delta(i+1, j+1) \leq A_\delta(i, j+1) + A_\delta(i+1, j)$$

This property can be proved as follows.

Theorem 2.3.6. *A_δ is a Monge matrix.*

Proof.

$$\begin{aligned} & A_\delta(i,j) + A_\delta(i+1,j+1) - A_\delta(i,j+1) - A_\delta(i+1,j) \\ = \; & \mathit{cost}(i,\delta) + S_{j+i_{c_0}-i} + \mathit{cost}(i+1,\delta) + S_{j+i_{c_0}-i} \\ & -\mathit{cost}(i,\delta) - S_{j+i_{c_0}-i+1} - \mathit{cost}(i+1,\delta) - S_{j+i_{c_0}-i-1} \\ = \; & p_{j+i_{c_0}-i} - p_{j+i_{c_0}-i+1} \; \leq \; 0 \end{aligned}$$

□

Definition 2.3.7. A matrix is *monotone* if for $1 \leq i_1 \leq i_2 \leq n$,$j(i_1) \leq j(i_2)$, where $j(i)$ is the column at which row i has its minimum value. If every submatrix is monotone, the matrix is *totally monotone.*

Every 2 x 2 submatrix of a Monge matrix is monotone, thus the Monge matrix is totally monotone. The algorithm described in [3] can determine all the $j(i)$ in a totally monotone matrix and has time complexity $O(n)$.

The dynamic programming algorithm is just to compute the cost of all the c_1-tuples by means of processing δ in lexicographic order. For example, let n be 5, to calculate $\mathit{cost}(2,3,3)$, we know $\delta = (2,3)$ and $j = 3$, and $A_\delta(i,3)$ need the value of $\mathit{cost}(i,\delta)$, $i \leq 2$, they are $(0,2,3), (1,2,3), (2,2,3)$. Since we process δ in lexicographic order, before $(2,3)$ is processed, $(0,2), (1,2), (2,2)$

must have been processed and the relative *cost* value has bee produced. After processing the last δ $(4, 4)$, our goal $cost(4, 4, 4)$ must have been completed.

There are n^{c_1-1} δ's to be processed, and each needs $O(n)$ time to find the $j(i)$. Note that entries of A_δ can be retrieved in constant time(by the argument in last paragraph). Thus the total time complexity is $O(n^{c_1})$.

Chapter 3

Improvements and Implementations

3.1 Digital Expansion Method

Digital expansion method can be viewed as a divide-and-conquer process that each time we partition the set of probabilities into two parts, then use the same partition way to deal with them individually, and assume the former partition is correct.

According to [15] digital expansion method generates code W where efficiency can be bounded as:

$$COST(W) - \frac{H}{\log X_0} \leq \frac{1 - p_1 - p_n}{\log X_0} + c_1 \tag{3.1}$$

When c_1 gets larger, $\log X_0$ becomes smaller, and performance degrades. Also observe that for skewed distribution, we can let p_1 and p_n be the largest two probabilities to improve the bound. We'll be back on this issue in chapter 4.

Due to its top-down nature, digital expansion method is adequate for adaptive coding. As to generalized Huffman method, if it's necessary to know where a symbol is located in a parsing tree, we need to build the whole parsing tree to acquire the exact path from root to the leaf(i.e. the codeword). However, as far as digital expansion method is concerned, we can get the codeword by simply deciding the path the symbol belongs to.

Figure 3.1 describes our adaptive digital expansion method algorithm. And Figure 3.2 illustrates how the algorithm works.

```
procedure $AdaptiveDEM(\ sym, low, high, sum\ )$
// sym: the symbol to be transmitted
// low: the index of the beginning of the section of the array
// high: the index of the end of the section of the array
// sum: the total sum of the probabilities indexed from low to high


// global variables
$X_0$: largest real root of equation $X_0^{-c_0} + X_0^{-c_1} = 1$
prob[$n$]: the array of probabilities

// initialization
$probsum := 0;$
$pivot := low;$

// only one probability, no need to split
If $low = high$
begin
        return;
end

//make pivot closest to the ideal index that separates the two piles
WHILE $probsum < sum\ *\ X_0^{-c_0}$
begin
        $probsum := probsum + prob[pivot];$
        $pivot := pivot + 1;$
end

// try to separate the two piles more precisely
IF $(probsum - sum) > (sum - (probsum - prob[pivot]))$
begin
        $probsum := probsum - prob[pivot];$
        $pivot := pivot - 1;$
end

IF $sym < pivot$
begin
        write bit 0 to stream;
        $AdaptiveDEM(sym, low, pivot - 1, probsum);$
else
        write bit 1 to stream;
        $AdaptiveDEM(sym, pivot, high, sum - probsum);$
end
```

Figure 3.1: Adaptive Digital Expansion Method

It can be seen that procedure AdaptiveDEM only expands the path regarding to the symbol in hand to be transmitted(as Figure 3.2 depicted). After transmitting the codeword, the probability model is modified. Then the next symbol is grabbed and processed just the same way, until data stream ends.

Also note that digital expansion method arranges the codeword by the appearance order of the probabilities. Take a simple example, for $c_0 = 1, c_1 = 2$, and $p_1 = 0.2, p_2 = 0.8$. Digital expansion method will arrange the left child of root node to the symbol with the probability 0.2, right child to the symbol with probability 0.8. Though the parsing tree is just the same as the optimal one, but the codeword mapping is inverse. We improve this by simply sorting the resulting codewords in the order of the probabilities. The symbol with higher probability shall be assigned a codeword with lower cost.

3.2 Generalized Huffman Method

3.2.1 Static Case

For original Huffman coding, it builds the parsing tree as follows. First, prepare a list of n nodes, each has its own probability of certain symbol. Then process the list as follows for $n-1$ times:

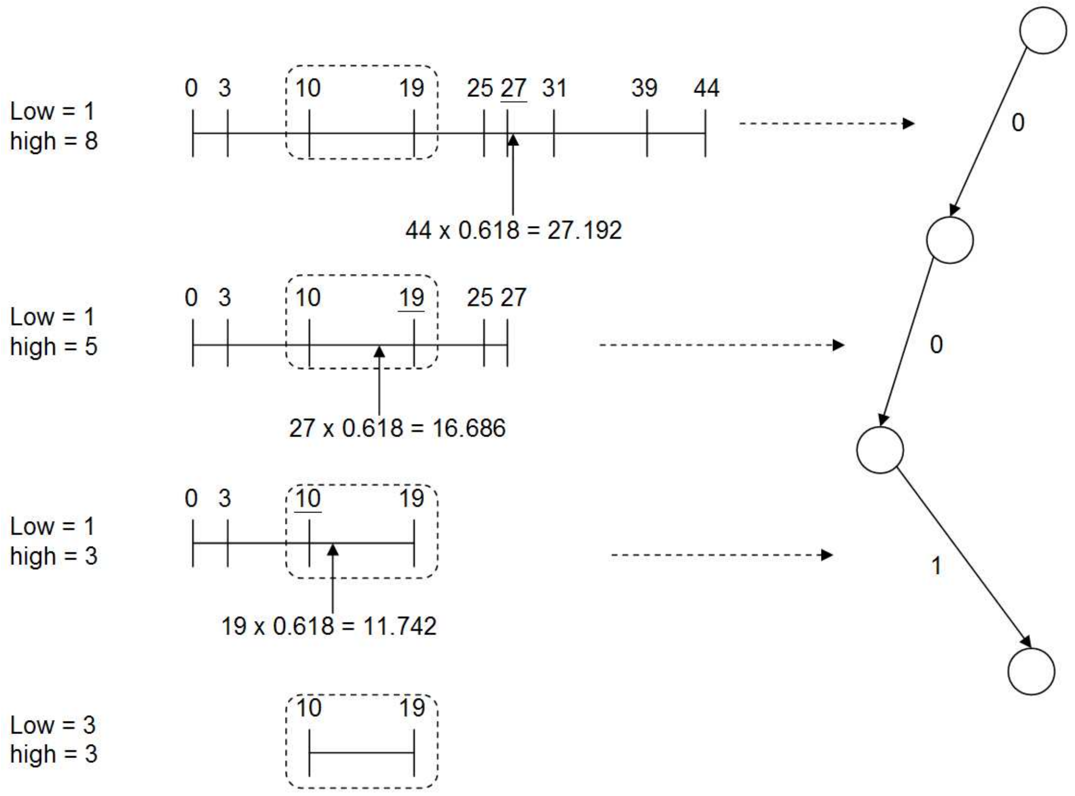
1. find two nodes, t_1 and t_2, with the least probabilities,
2. create a node as their parent node t_p,
3. let the probability of t_p be the sum of probabilities of t_1 and t_2,
4. delete t_1 and t_2.

The only node remained is the root node of the parsing tree.

The first step needs $O(\log n)$ searching time if we use a heap structure to manipulate the list (in fact, no better data structure exists in this case), and the next three steps need constant time. Thus to build a parsing tree, totally $O(n \log n)$ is required.

As for the generalized Huffman method, t_1 is the node with the least probability, and t_2 is the one with probability closest to t_1's probability multiply $X_0^{c_1-c_0}$. To find t_2 we need a different data structure instead of array. If we use arrays to implement generalized Huffman coding, we'll need $O(n)$ time complexity to search for t_2.

We choose binary search tree as the data structure. We establish a binary search tree such that the keys of the nodes are their probabilities. We need the following binary search tree manipulation routines:



1. $c_0 = 1, c_1 = 2, X_0^{-c_0} = 0.618$

2. 8 symbols occur for $\{3, 7, 9, 6, 2, 4, 8, 5\}$ times respectively, and the symbol to be transmitted is the one occurring for 9 times(its section of probability is circled by dashed line).

3. In each stage of separation, if the symbol to be transmitted is in the left pile, go down left link(i.e. bit 0), otherwise, go down right link(i.e. bit 1). The accumulated sum closet to $X_0^{-c_0}$ is underlined.

4. The final codeword for it is 001. At the last stage, only one symbol remains, thus end the expanding.

Figure 3.2: top-down expanding

1. $Insert(t, T)$: insert node t into the binary search tree T.

2. $Delete(t, T)$: delete node t from the binary search tree T.

3. $FindMin(t, T)$: in the subtree of T, which root node is t, find the node with the smallest key then return it.

4. $FindNearest(k, T)$: find the node with key value nearest to k, then return it.

The algorithm is shown in Figure 3.3.

All the binary search tree routines need $O(d)$ time, where d is the depth of the tree. Therefore our algorithm needs $O(n \cdot d) = O(n^2)$ time to build the parsing tree.

Note that the binary search tree can be further improved to be balanced, since the nodes added are inclined to be monotone increasing, especially when there are fewer nodes remain in the tree-building phase. We know that for a monotone increasing sequence the binary search tree may degrade to have search time $O(n)$. A balanced binary search tree(e.g. red-black tree[5], skip lists[16],. . . etc) can avoid worst case, achieve $O(\log n)$ time complexity and guarantee the overall time complexity $O(n \log n)$. But the programming complexity and overhead shall be cautiously handled.

The code generated by the parsing tree may have a problem that a symbol with smaller probability does not necessarily own the codeword with the greater cost. Like digital expansion method, we can simply sort the symbol-codeword mapping by means of the costs of the codewords. Since it can be observed that the mapping is already nearly sorted, we choose insertion sort and achieve $O(n)$ time complexity[17].

3.2.2 Adaptive Case

We also develop a adaptive algorithm using generalized Huffman coding. Since generalized Huffman method is a bottom-up approach, we can't use the way we modify the static digital expansion method to reduce the amount of computation. And the tree-building phase is very time-consuming actually. Thus it's natural to think of some 'lazy' way building the parsing tree for less times. Since the generalized Huffman method tries to approximate the optimal parsing tree by 'mimicking' the characteristic of zero-redundancy coding, we develop our adaptive algorithm as follows. When a parsing tree is temporarily built, we give each pair of children a value *origin*, that stores the

```
// data structure prepared
prob[$n * 2$]: array for internal and external nodes
parent[$n * 2$]: array for storing parent node
lchild[$n$]: array for storing left child node
rchild[$n$]: array for storing right child node
T: empty binary search tree

// Initialization
For $i := 1$ to $n$
begin
    $Insert(i, T)$;
end
$t_p := n + 1$;

// tree-building phase
While $t_p < n * 2$ // there are more than one node in $T$
begin
    $t_1 := FindMin(root\ of\ T, T)$;
    $t_2 := FindNearest(t_1, T)$;

    parent[$t_1$] := $t_p$;
    parent[$t_2$] := $t_p$;
    rchild[$t_p$] := $t_1$;
    lchild[$t_p$] := $t_2$;

    prob[$t_p$] := prob[$t_1$] + prob[$t_2$];

    $Insert(t_p, T)$;
    $Delete(t_1, T)$;
    $Delete(t_2, T)$;

    $t_p := t_p + 1$;
end
```

Figure 3.3: Generalized Huffman coding

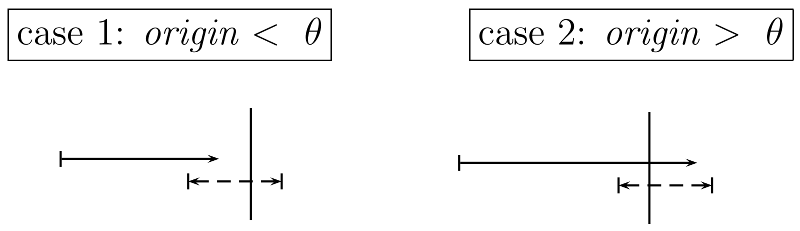


Figure 3.4: Deciding Range

ratio of their probabilities. *origin* may be larger than the ideal ratio $X_0^{c_1-c_0}$ or smaller. While we use this parsing tree to process the next symbol to be transmitted, the model of probability is changed. If the ratio of probabilities of this symbol and its sibling is out of range , we need to rebuild the parsing tree. Not only the ratio of the pair of leaf nodes are checked, but also all the ratio of the ancestors along the path to the root and their siblings are checked. Let $\theta = X_0^{c_1-c_0}$. The ratio r must be in the range:

$$origin \begin{cases} < \theta \ : \ (origin \ / \ r_2) \ < \ r \ < \ (\theta \ * \ r_1) \\ \ o/w \ : \ (\theta \ / \ r_1) \ < \ r \ < \ (origin \ * \ r_2) \end{cases}$$

where r_1 and r_2 are floating point numbers that assist to decide the range of r.

If the original ratio has been larger than the ideal θ, we allow the ratio to be larger than $origin * r_2$(let *origin* decide the upper bound). But the ratio may also become smaller, and we require it to be at least $\theta * r1$(let θ decide the low bound). We can do similarly for the ratio smaller than θ.

Here we explain why we store *origin* values. In generalized Huffman coding, sometimes the ratios of certain pairs of siblings could be very far from the ideal θ, and we can't avoid it. If we decide the range only by θ, very soon we'll need to rebuild the parsing tree and we'll get a similar one. This won't help a lot for reducing redundancy but takes much time for rebuilding the parsing tree. Hence we shall consider *origin* too, for deciding the ranges of ratios. We show how the above mechanism works in Figure 3.4 and Figure 3.5.

In Figure 3.5, suppose $c_0 = 1$, $c_1 = 2$, hence $\theta = 1.618$. Four symbols so far have been transmitted $\{1, 2, 3, 10\}$ times, respectively. We call it a

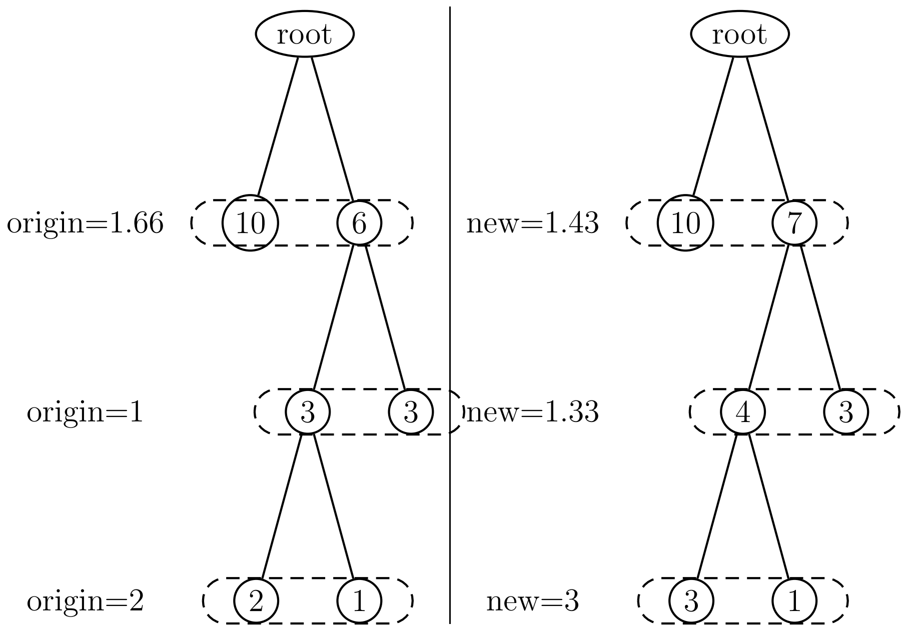


Figure 3.5: Adaptively Rebuild Parsing Tree

weight distribution. In the left part of the figure, a parsing tree has been built by using the static generalized Huffman method. From the lowest level, the pairs of siblings have ratio $\{2, 1, 1.66\}$. They are stored as *origin*.

The right part of the figure shows that when the symbol that appears right then is the one with *weight* 2. Now the ratios have been changed to $\{3, 1.33, 1.43\}$, respectively. If any of the ratios is out of the range which depends on *origin* and θ, we need to rebuild the parsing tree. If the ratios are all still in the ranges they are constrained, the parsing tree won't be modified and will be used to encode the next symbol.

When a new symbol is added, the parsing tree is forced to be . Note that after rebuilding the parsing tree could be just the same one as before. The remaining problem is: what r_1 and r_2 shall we choose? If r_1 is too large, the ratios could become far from the ideal value θ, and the resulting parsing tree can't keep up with the probability distribution so far and thus result in more redundancy. Or if r_1 is too small, the frequency of rebuilding could get higher and slow down the whole coding process. Similarly, if the chosen r_2 is too large, redundancy is produced. If r_2 is too small, frequently rebuilding causes a waste of time.

In our example, the original ratio of the lowest pair of siblings is 2(left side), and by now the ratio is 3(right side), larger than *origin*(case 1 in Figure 3.4). If $r_1 = 1.2$, $3 > 1.2*2$ (i.e., out of range), then the parsing tree is rebuilt. If $r_1 = 1.6$, $3 < 1.6 * 2$ (i.e, still in the range), then we do nothing.

In chapter 4, we'll show experimental results about the effects of choosing different r_1 and r_2.

3.3 Dynamic Programming Approach

Although dynamic programming algorithm is such an elegant approach which runs in polynomial time, we find that the enormous space it uses is really a problem worth further studying.

The spirit of dynamic programming is cutting the main problem into small sub-problems and solving them. The solutions of sub-problems can be easily used to establish the solution of main problem. Any solution may be used at some time, we can't delete it to save the space. Thus there is no much room for us to think of some method to reduce the space complexity $O(n^{c_1})$.

Assume a system equipped with 4GB memory, and each entry in the *cost* array need 4 bytes, we do the following analysis. If there are 256 source symbols(byte-based), the system can afford $c_1 \leq 3$ cases. As to a typical text file, there are around 80 source symbols, the system can afford only $c_1 \leq 4$ cases. Even for $n = 27$(English letter plus space), c_1 can only be at most 6. Before time appears to be a problem, memory has been exhausted.

Since the entries in the tuples must be monotonely increasing, by simple combinatorical argument, there are only $\binom{n+c_1-1}{c_1}$ entries of *cost* array used, other entries are useless. This is similar to the problem we illustrated in Figure 3.6. Thus the space usage ratio is $\frac{\binom{n+c_1-1}{c_1}}{n^{c_1}}$. When c_1 changes from c to $c+1$, the space usage ratio also changes:

$$\frac{\binom{n+c}{c+1}}{n^{c+1}} \Bigg/ \frac{\binom{n+c-1}{c}}{n^{c}} = \frac{n+c}{n(c+1)} < 1(\because n \geq 1, c > 1)$$

The space usage ratio keeps decreasing. Also note when n is much larger than c_1, the space usage ratio is approximately $\frac{1}{c_1!}$.

Also we can deduce that over a binary channel with c_0, c_1 fixed, more source symbols(i.e. $n \uparrow$) results in worse space usage.

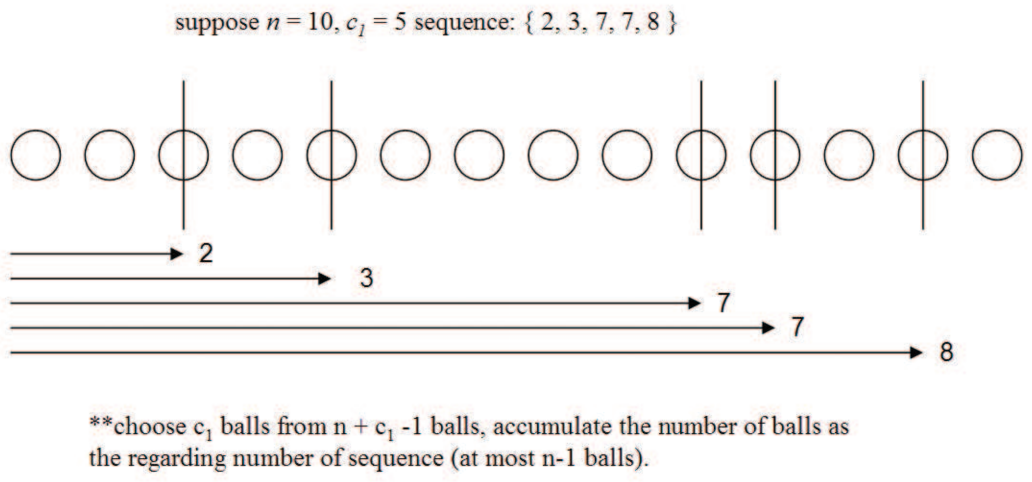

suppose $n = 10$, $c_1 = 5$ sequence: { 2, 3, 7, 7, 8 }
2
3
7
7
8
**choose c_1 balls from $n + c_1$ -1 balls, accumulate the number of balls as the regarding number of sequence (at most n-1 balls).

Figure 3.6: analogy of choosing balls

Here we present an easy way to fully use the space for storing the *cost* array. First we initialize the *cost* array with size $\binom{n+c_1-1}{c_1}$. Again combinatorics offer the fact that for a c_1-tuple $(d_0, \ldots, d_{c_1-1})$, we can give it an index number which depends on n and c_1. Let $i_{n,c_1}(s_{begin}, s_{end})$ be the number of the monotonely increasing sequences between sequences s_{begin} and s_{end}, and each has c_1 entries with value from 0 to $n-1$. For example, $i_{3,2}(01, 12) = 4$, since there are 4 suitable sequences $01, 02, 11, 12$.

We can calculate the index number by dealing with each digit separately. For example, to calculate $i_{10,4}(0000, 6779)$, we have:

$$\begin{aligned} i_{10,4}(0000, 6779) &= i_{10,4}(0000, 6665) + i_{10,4}(6666, 6779) \\ &= [i_{10,4}(0000, 9999) - i_{4,4}(0000, 3333)] + i_{4,3}(000, 113) \end{aligned}$$

The sequences after 6665 have entries smaller than $10 - 6 = 4$. It's easy to calculate $i_{10,4}(0000, 9999) - i_{4,4}(0000, 3333) = \binom{10+4-1}{4} - \binom{4+4-1}{4} = 715 - 35 = 680$. Similarly, we can replace $i_{4,3}(000, 113)$ with

$$\begin{aligned} i_{4,3}(000, 111) + i_{3,2}(00, 02) &= [i_{4,3}(000, 333) - i_{3,3}(000, 222)] + i_{3,2}(00, 02) \\ &= \binom{4+3-1}{3} - \binom{3+3-1}{3} + i_{3,2}(00, 02) \\ &= 20 - 10 + 3 = 13 \end{aligned}$$

Thus $i_{10,4}(0000, 6779) = 680 + 13 = 693$.

Every time we need to retrieve an entry in the array, we compute the index number. Since c_1 is being viewed as a constant, computing this index number is a feasible procedure that won't interfere the $O(n^{c_1})$ time complexity.

For another interesting situation, c_1 may be much larger than n. Our array-indexing method can trim the space used from intractable size to extremely small amount. However, if c_1 is so large, it's more realistic to try all the parsing tree. Since there are only $\frac{1}{n}\binom{2n-2}{n-1}$ (($n-1$)th Catalan number) n-leaf full binary trees.

Also note that our array-indexing method only offers marginal improvement. It makes more cases possible to run on modern computers, but the breakthrough in space efficiency still relies on theoretical result.

Chapter 4

Experimental Results

In this chapter we show our experimental results. Our test data includes the Canterbury Corpus[6] and several *de facto* standard compression evaluation files.

The 8 files listed below are to be compressed:

1. alice.txt: a standard text file from Canterbury corpus.
2. fields.c: a source code file from Canterbury corpus.
3. ptt5: a fax image file from Canterbury corpus.
4. sum: an executable file from Canterbury corpus.
5. bible.txt: a large text file from Canterbury corpus.
6. E.coli.txt: a large file from Canterbury corpus, with only 4 symbols(A,T,C,G).
7. lena.tiff: *de facto* standard image files
8. peppers.tiff: *de facto* standard image files

Our experiments run on a machine equipped with the following setup:

- CPU: Intel Pentium 4 3.2GHz processor(800MHz FSB)
- Memory: 1GB RAM(Dual DDR400)
- Operating System: Windows Server 2003

4.1 Adaptive Digital Expansion Method

In this section we adjust r_1 and r_2(as described in section 3.2.2) and run experiments to find the tradeoff between compression ratio and performance.

We have tested 6 cases: $r_1 = r_2 = 1, 1.1, 1.2, 1.5, 2.0, 3.0$. Ideally, higher value means wider range, thus less times for rebuilding the parsing tree and the redundancy increases(see section 3.2.2). If $r_1 = r_2 = 1$, with high probability the incoming symbols will very soon make the ratios out of the range. This will take lots of time to rebuild the parsing tree and make the adaptive coding very slow. Thus we consider the other cases. From the results of chapter 2, the redundancy percentage rp is defined as:

$$rp = \frac{COST(W) - \frac{H}{\log X_0}}{\frac{H}{\log X_0}} \tag{4.1}$$

The code W we generate can't be better than the ideal lower bound $\frac{H}{\log X_0}$. Smaller rp means lower redundancy.

Ideally, rp is in inverse proportion to the processing time. However, in our experiments only 4 files obey this. We only show these 4 files in Figure 4.1. Although other files are not shown, in fact they are inclined to be that rp is in proportion to the processing time, but in a much irregular way.

We can see that when r_1 and r_2 are large enough, the time spent won't be reduced. If throughput is in consideration, 2.0 case is suitable. By now, we focus on rp so we choose 1.2 case in the following experiment. Also note that sometimes even the larger values are chosen, the rp value is not improved. Since the adaptive generalized Huffman method is just to "approximate", the parsing trees obtained are not guaranteed to be the optimal ones. Sometimes fewer rebuilding of parsing tree would get better results, which is contrary to the expectation.

4.2 Comparison of Two adaptive Algorithms

In this section we run experiments for our two adaptive algorithms. We choose 3 cases, $c_0 : c_1 = \{\ 1:2,\ 1:5,\ 1:10\ \}$. The results are shown in Figure 4.2 - 4.4.

In the $1:2$ case, note that the files *ptt5* and *E.coli.* have very skewed distributions(87.12% of the file are the symbol '0'). For a skewed distribution, any minor difference from the optimal parsing tree may cause severe cost. For the approximation method, by chance it could build a parsing tree with high cost. The file *E.coli* has only 4 symbols. Thus the parsing tree is very

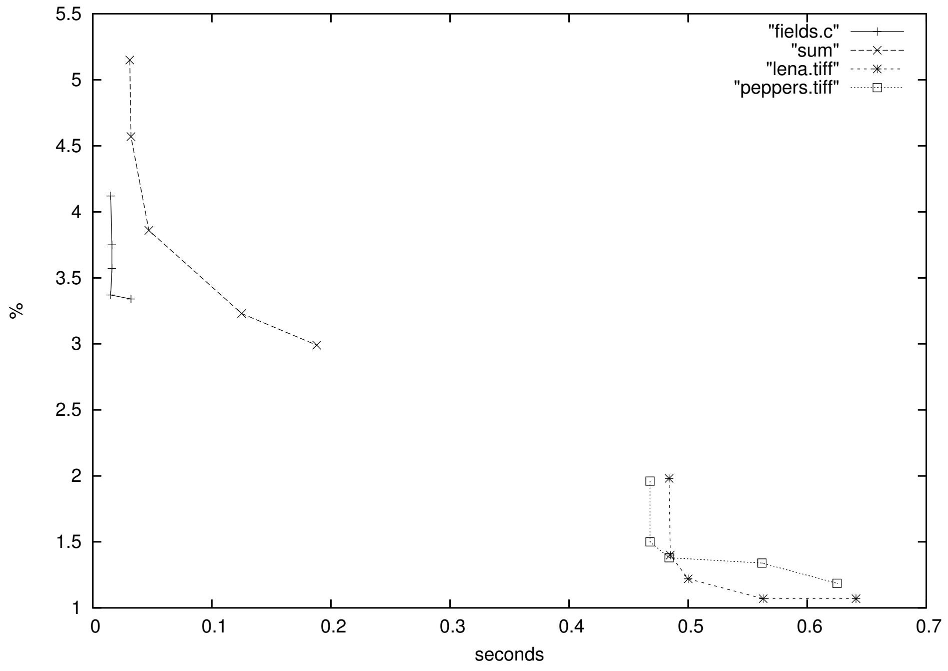


Figure 4.1: $r_1 = r_2 = 1.1, 1.2, 1.5, 2.0, 3.0$

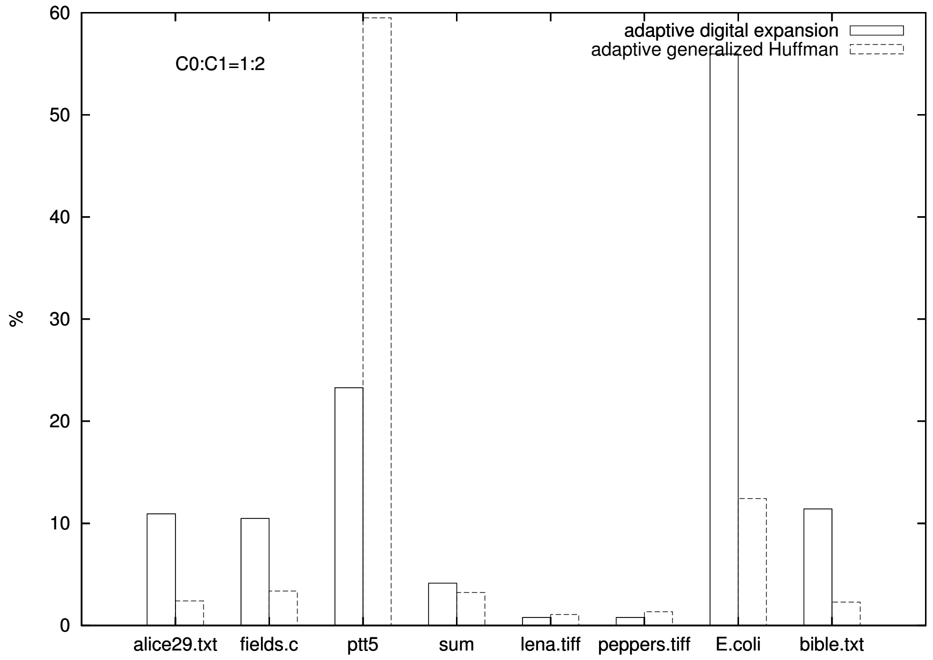


Figure 4.2: performance comparison of adaptive algorithms - 1:2

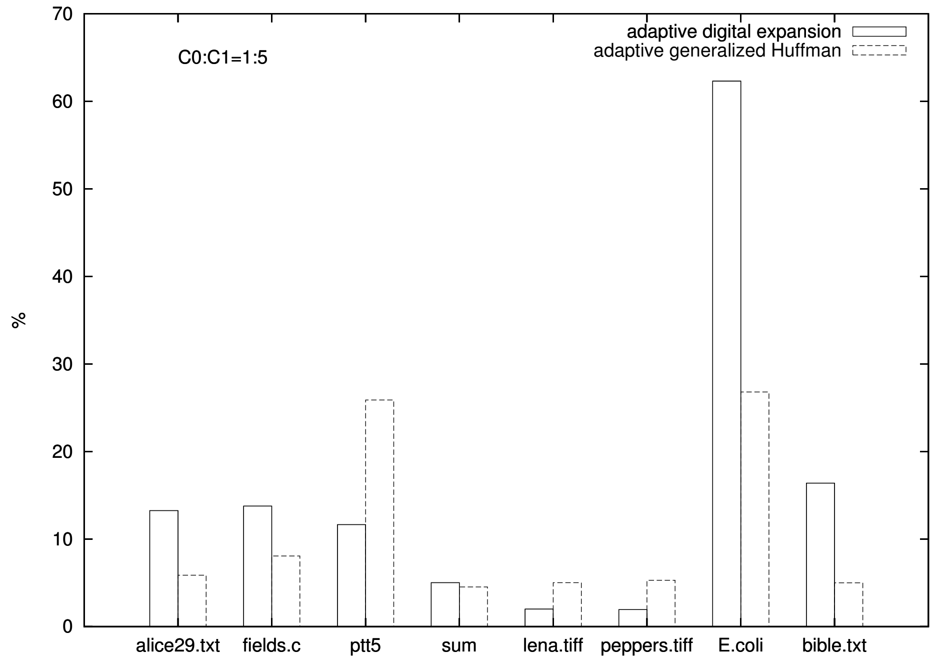


Figure 4.3: Performance comparison of adaptive algorithms - 1:5

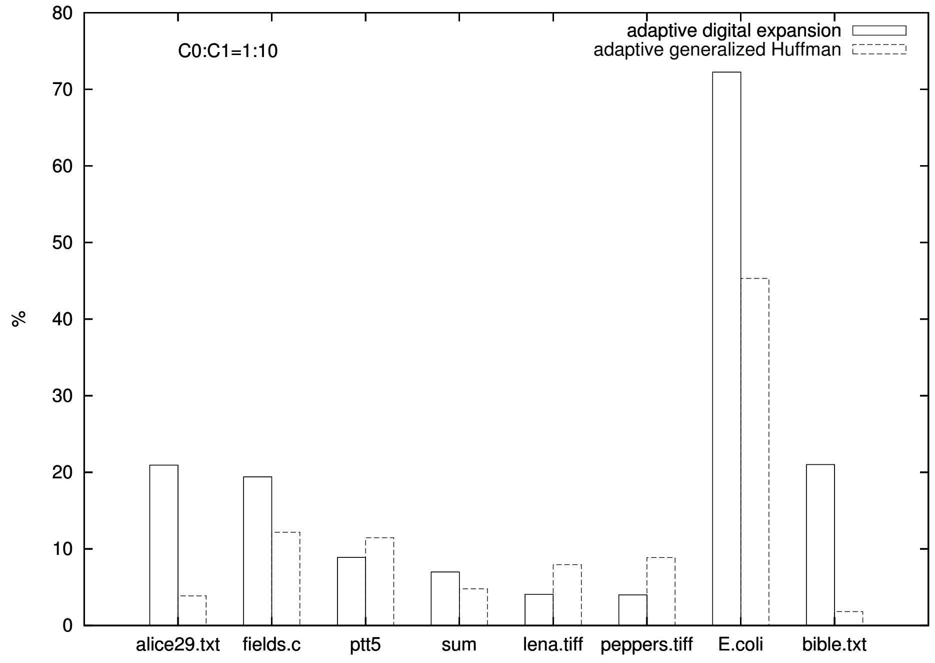


Figure 4.4: Performance comparison of adaptive algorithms - 1:10

small and there are only a few possibilities. Similarly, the approximation method could build a tree with a slight difference with the optimal one and has cost far from optimal.

In the cases $1:5$ and $1:10$, results of *ptt5* have been improved, but *E.coli* still gets worse. Since there are only 4 symbols in *E.coli*, we suggest using exhaustive search for all binary trees and it will be more precise and faster.

Compare the two adaptive algorithms, adaptive generalized Huffman coding performs better for most situations. However, for images files, the redundancy percentage of adaptive digital expansion method is apparently smaller, approximately one-half. That's because image files have high "locality", and adaptive digital expansion method updates its model of probability every time a symbol is transmitted. But for adaptive generalized Huffman method, its "lazy" update method sometimes can't keep up with the locality of data. For the text files *alice29.txt* and *bible.txt*, adaptive generalized Huffman method is more suitable.

4.3 Comparison with Static Algorithms

In this section we compare the original two static algorithms. In both algorithms, after the parsing trees are built, the codeword mapping are sorted to guarantee that symbols with lower probability will be mapped to higher cost codewords.

Again we test 3 cases:$\{\ 1:2,\ 1:5,\ 1:10\ \}$. We show the results in Figure 4.5 - 4.7. In $1:2$ case, performance of generalized Huffman method performs better. However, in the other cases, performance of digital expansion method becomes better than generalized Huffman method. Generally, these two static methods have codewords with redundancy lower than 5%, rarely high than 10%.

Note that when $c_0 : c_1 = 1:2$, the optimal parsing tree for the file *ptt5* has $rp = 19.12\%$(we calculate this by the dynamic programming approach). Thus indeed static digital expansion method has achieved near-optimal result. And when $c_0 : c_1 = 1:10$, the optimal parsing tree for the file *E.coli* has $rp = 16.48\%$. Both the static algorithms find the optimal code.

4.4 Dynamic Programming Approach

As described in section 3.3, to acquire the optimal parsing tree, we need very much computation power and memory space. For the test files, we have the following results. For every file containing n symbols, if $c_1 \leq c$, then we are

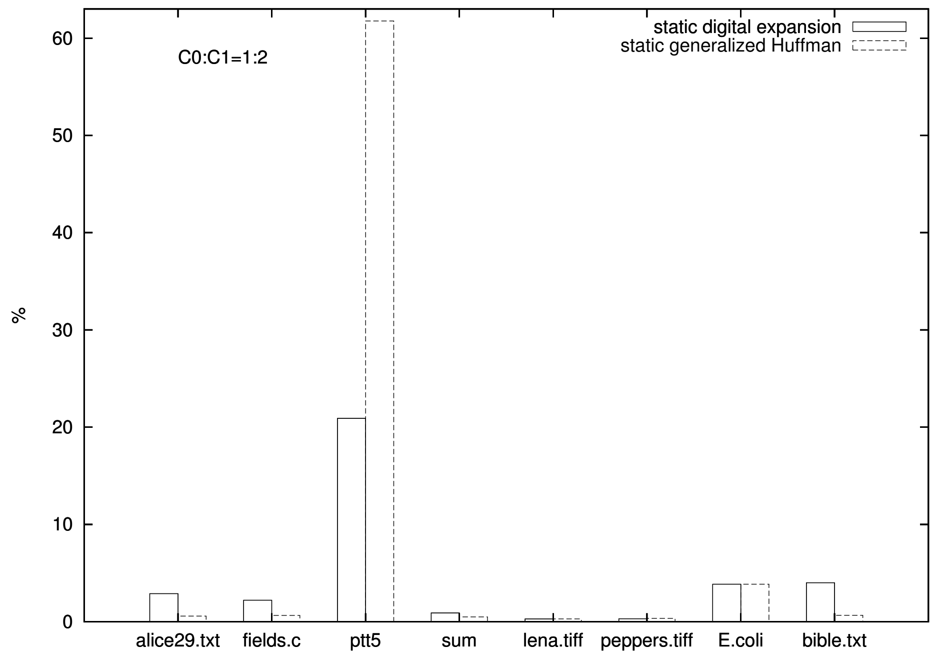


Figure 4.5: Performance comparison of static algorithms - 1:2

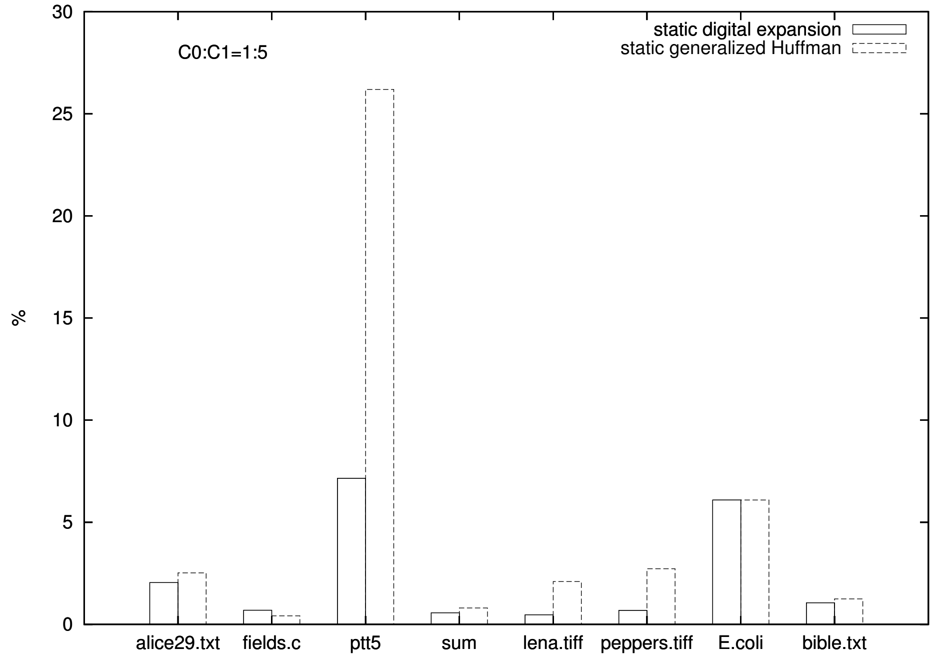


Figure 4.6: Performance comparison of static algorithms - 1:5

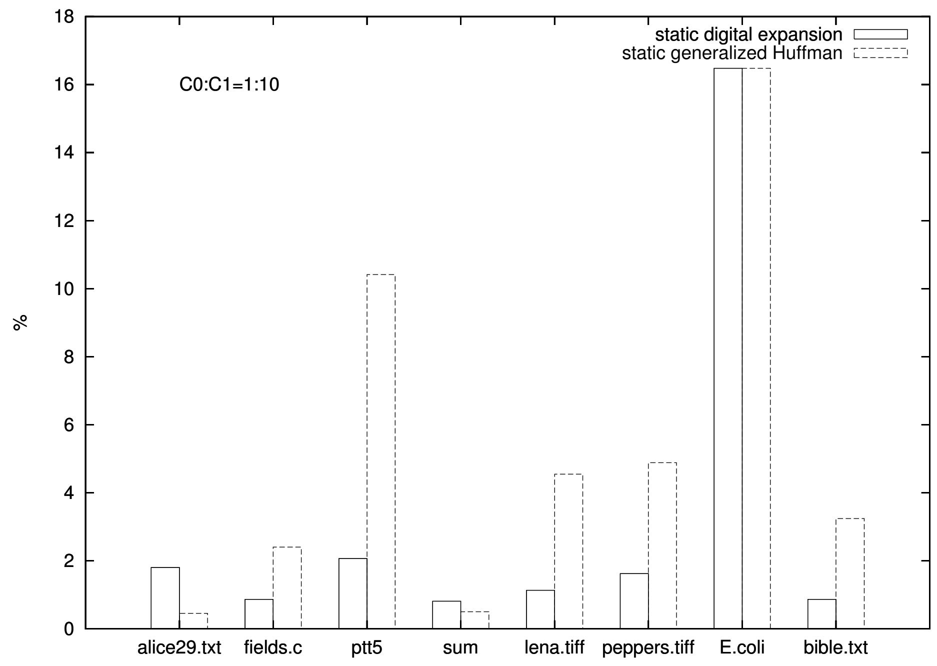


Figure 4.7: Performance comparison of static algorithms - 1:10

able to acquire the minimum-redundancy with a tree-based coding.

	n	c	rp
alice29.txt	74	5	0.11%
fields.c	90	5	0.09%
ptt5	159	4	7.57%
sum	255	4	0.22%
lena.tiff	256	4	0.11%
peppers.tiff	255	4	0.26%
bible.txt	63	6	0.12%

Generally, text files have fewer symbols, therefore c_1 can be a little larger. We also compress an English text file with 27 symbols(its distribution was as mentioned in [9]), c_1 can be at most 9, $rp = 0.18\%$.

Chapter 5

Conclusion

Our adaptive generalized Huffman coding offers the function for adjusting the parameter r_1 and r_2. And its performance is better than adaptive digital expansion method. In static cases, when c_1 becomes large, digital expansion method is better.

Digital expansion method seems to be very suitable for coding image files. When c_1 is small, generalized Huffman coding can achieve very low rp for text files.

Also note that for files with very skewed distribution sometimes generalized Huffman coding has very high rp. Digital expansion method has an upper bound as mentioned in section 3.1.

When optimal solution is needed, dynamic programming approach can be used but very limited. When n is small(< 15), we suggest exhaustive search.

When c_1 is very large, both static digital expansion method and generalized Huffman method perform well, but not in the adaptive case. [9] have provided some other solutions for large c_1, but considered only static case.

Bibliography

[1] Julia Abrahams, "Code and parse trees for lossless source encoding," *IEEE Communications in Information and Systems,* 1(2):113-146, April 2001.

[2] Julia Abrahams and Marc J. Lipman, "Zero-redundancy coding for unequal code symbol costs,"*IEEE Transaction on Information Theory,* 38(5):1583-1586, September 1992.

[3] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, Robert Wilber, "Geometric applications of a matrix searching algorithm,"*Proc. Annual Symposium on Computational Geometry,* 2:285-292, 1986.

[4] D, Altenkamp and K. Mehlhorn, "Codes: Unequal probabilities, unequal letter costs," *Journal of the ACM,* 27(3):412-427, July 1980.

[5] Arne Andersson,"Balanced search trees made simple," *Sorkshop on Algorithms and Data Structures,* 60-71, Springer Verlag, 1993.

[6] Ross Arnold and Tim Bell, "A corpus for the evaluation of lossless compres- sion algorithms," *Data Compression Conference, IEEE Computer Society Press,* 201-210, 1997.

[7] Phil Bradford, Mordecai J.Golin, Lawrence L. Larmore, and Wojciech rytter, "Optimal prefix-free codes for unequal letter costs: dynamic programing with the Monge property," *Proc. Sixth European Symposium on Algorithms,* 43-54, 1998.

[8] Shlomi Dolev, Ephraim Korach, Dmitry Yukelson, "The sound of silence: guuessing games of saving energy in mobile environment," INFOCOM '99. Eighteenth annual joint conference of the IEEE Computer and Communications Societies. Proc. IEEE, 2:768 - 775, March 1999.

[9] E.N.Gilbert, "Coding with digits of unequal costs," *IEEE Transaction on Information Theory,* 41:596-600, 1995.

[10] Mordecai J. Golin, Claire Kenyon, and Neal E. Young, "Huffman coding with unequal letter costs," Proc. 34th annual ACM symposium on Theory of computing, 785-791, 2002.

[11] Mordecai J. Golin, Günter Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs," *IEEE Transactions on Information Theory,* 44(5):1770-1781, September 1998.

[12] David A. Huffman, “A method for the construction of minimum redundancy codes,” *Proc. IRE*, 10:1098-1101, September 1952.

[13] Richard Karp, “Minimum-redundancy coding for the discrete noiseless channel,” *IEEE Transactions on Information Theory,* 7(1):27-38, January 1961.

[14] R. S. Marcus, “Discreate noiseless coding,” *M. S. Thesis, MIT,* Elec Eng. Dept, 1957.

[15] K. Mehlhorn, “An efficient algorithm for constructing nearly optimal prefix codes,” *IEEE Transaction on Information Theory,* 26:513-517, September 1980.

[16] J. Ian Munro, Thomas Papadakis, Robert Sedgewick,“Deterministic skip lists, ” *Proc. 3rd ACM-SIAM Symp on Discrete Algorithm*, 367-375, January 1992.

[17] Robert Sedgewick, *Algorithm in C,* Addison-Wesley, 1990.

[18] C. E. Shannon, “A mathemetical theory of communication,” Bell System Technical Journal, 27:379-423,623-656, July, October, 1948.