Homework #1 Report

Multimedia Data Compression

EE669 2015Spring

Name: Shanglin YANG

ID: 3795329308

Email: Shangliy@usc.edu

Content

Problem 1: Written Questions

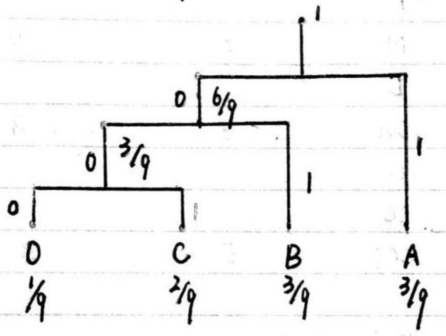
1.1 Huffman Coding

1. a) $E = -\sum_{i=0}^{N} P(S_i) \cdot \log_2 P(S_i) = -\left(\frac{3}{9} \times \log_2 3/9 + 3/9 \log_2 3/9 + 2/9 \log_2 \frac{2}{9} + 1/9 \log_2 1/9\right)$

$= -\left(3/9 \times (-1.58496) + 3/9 \times (-1.58496) + 2/9(-2.169925) + 1/9(-3.169925)\right)$

$\doteq 1.8911$

b) $P(A) = 3/9 \quad P(B) = 3/9 \quad P(C) = 2/9 \quad P(D) = 1/9$



A: 1
B: 01
C: 001
D: 000

c) fixed $n = \log_2 4 = 2$
Huffman $n = \sum_{i=1}^{4} L(S_i) \cdot P(S_i) = 1/9 \times 3 + 2/9 \times 3 + 3/9 \times 2 + 3/9 \times 1$

$= 1/3 + 2/3 + 1 = 2$

d) $R = N - E = 2 - 1.8911 \doteq 0.1089$

e) ABDCABBBCADB CABCAA
1 01 000 001 1 01 01 01 001 1 000 01 001 1 01 001 1 1
ABOCABBBC ADBCAB CAA

1.2 Lempel-Ziv Coding

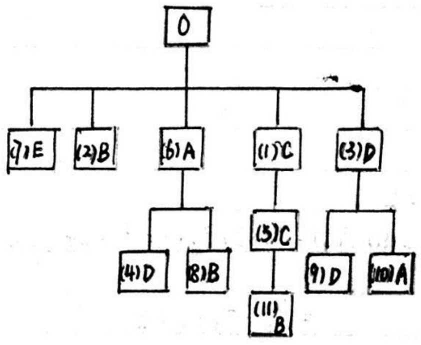
2、

a)

Dictionary		Input	Left	Output Phrase	Output Character
0	Null	C	CC	1	C
1	C	A	A	0	A
2	B	E	E	0	E
3	D	A	A		
4	AD	B	AB	6	B
5	CC	D	D		
6	A	D	DD	3	D
7	E	D	D		
8	AB	A	DA	3	A
9	DA	C	C		
10	DA	C	CC		
11	CCB	B	CCB	5	B

Code： (1, C) (0,A) (0,E) (6,B) (3,D)(3,A) (5,B)

(b)



1.3 Arithmetic Coding

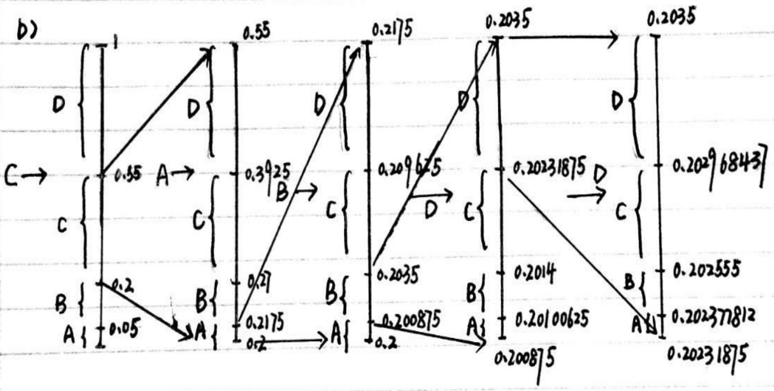
3、 $S = \{A, B, C, D\}$, $\{0.05, 0.15, 0.35, 0.45\}$

a) In general, unlike huffman code which assign variable and integer length code to each block, arithmetic code is a non-block code and more efficient than Huffman codin

For a set of symbol with skewed probability, considering the huffman always use integer length, the code redudancy could be big.

However, the arithmetic take the whole set as the object, and achieve the theoretical optimal result for the whole set. Huffman coding is only close to an arithemaac code when the probabilities of each symbol are the negative power of 2.

b)



we can use any ~~range~~ number within the range of $[0.20296843, 0.2035]$ to represent "CABDD", in general, we chose central value. like. 0.2032.

c. $T = (T - B_L) / (B_U - B_L)$

(1) $T \in [0.2, 0.55] \therefore \Rightarrow$ 'c', & $T = \dfrac{0.4507153 - 0.2}{0.35} = 0.71643$

(2) $T \in [0.55, 1] \therefore \Rightarrow$ 'D' & $T = \dfrac{T - 0.55}{0.45} = 0.36986$

(3) $T \in [0.2, 0.55] \therefore \Rightarrow$ 'C' & $T = \dfrac{T - 0.2}{0.35} = 0.4853$

(4) $T \in [0.2, 0.55] \therefore \Rightarrow$ 'C' & $T = \dfrac{T - 0.2}{0.35} = 0.8151$

(5) $T \in [0.55, 1] \therefore \Rightarrow$ 'D'

$\therefore$ String is 'CDCCD'

Problem 2: Entropy Coding

2.1 Abstract and Motivation

Entropy coding is an important step in multimedia data compression. It is a lossless compression scheme which transforms symbols into binary codes. It assigns each symbol with a unique prefix-free code and the length of each code depends on the probability of the corresponding symbol. The symbol with larger probability is assigned with a shorter code. [1] In this section, I will implement two famous encoding methods: Shannon-Fano coding, Huffman and adaptive Huffman coding and discuss their performance.

2.2 Approach and Procedures

2.2.1 Shannon-Fano Coding

A. Concept and procedure

In the field of data compression, Shannon–Fano coding, named after Claude Shannon and Robert Fano, is a technique for constructing a prefix code based on a set of symbols and their probabilities (estimated or measured). [i]

The procedure flowchart of the Shannon Fano coding is shown in Figure 2.1.At first, the symbols are arranged in order from most probable to least probable, and then divide symbols into two sets whose total probabilities are as close as possible to being equal. All symbols then have the first digits of their codes assigned; symbols in the first set receive "0" and symbols in the second set receive "1". As long as any sets with more than one member remain, the same process is repeated on those sets, to determine successive digits of their codes. When a set has been reduced to one symbol this means the symbol's code is complete and will not form the prefix of any other symbol's code.

To calculate the entropy, we use the equation:

$$H = -\sum_{k=1}^{L} P(S_k)\log_2 P(S_k) \qquad (2.1)$$

where $P(S_k)$ is the probability of Symbol S_k .

B. Data structure and Key function

(1) To clearly represent the node of and generate the code , construct the structure of the node and code for each leaf node.

```
class SYMBOL    //the SHANNON node construction
{
    public:
    SYMBOL* left;   //the left side
    SYMBOL* right;  //the right side
    unsigned int pro;  //the weight of the root
    unsigned char name; //the symbol name of the root
    SYMBOL() { left = right = NULL; pro = 0; name = '\0';  };  //initial the
```

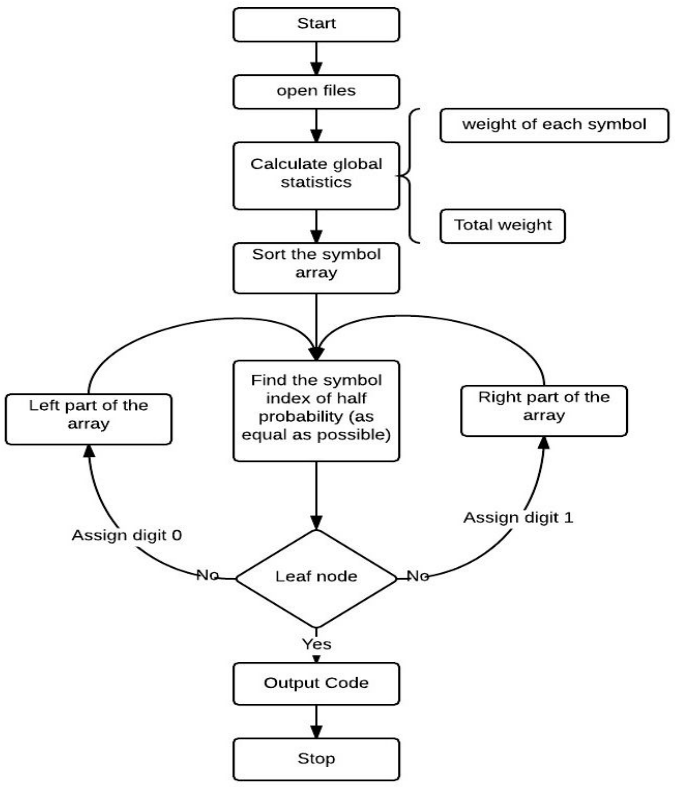


Figure2.1 Procedure of Shannon-Fano Coding

```
    new structure
    SYMBOL(SYMBOL* l, SYMBOL* r, int p, char n) { left = l; right = r;
     pro = p;    name = n; } //send data to the construction
    ~SYMBOL() { delete left; delete right; } //delete the construction
};

class CODSYM
{
    public:
    SH_Code code;       // Using the vector to represent the code
    unsigned char name;
    CODSYM() { name = NULL; };
}CODE[256];
```

(2)Most important part of the algorithm is to sort and find the index which make to sum of probability(weight) of right and left are as most as equal.To accomplish that , I use the **sort_ARR()** and **Root_find()** function . After building the tree, I use the **shannon_generate_code()** to generate the function.

```
void sort_ARR(SYMVector &pro_data, int L)
void Root_find(SYMBOL &root, double *sum_p, int le, int ri)
void shannon_generate_code(SYMBOL &root, SH_Code&scode)
```

2.2.2 Huffman Coding

A. Concept and procedure

Huffman coding is developed from Shannon-Fano coding. And it is also a lossless compression scheme based on the frequency of each symbol. the Huffman algorithm works from leaves to the root and make sure unique prefix and optimal result.[ii]

The procedure flowchart of the algorithm is is shown in Figure 2.2. At first, calculate the global statistics, including the weights of each symbol, the total weight. Then take all symbol as leaf node and find two nodes with the smallest weight. Combine two node as new node whose weight equal to the sum of the two node.Drop two old and new node to the array,repeat the find process until reach the root of the binary tree. In Encode process,do it from the root to the leaves of the binary tree. Assign the node which is the left child of its parent digit 0 and the one which is the right child of its parent digit 1. When we reach the leaves, the code for each symbol is generated.

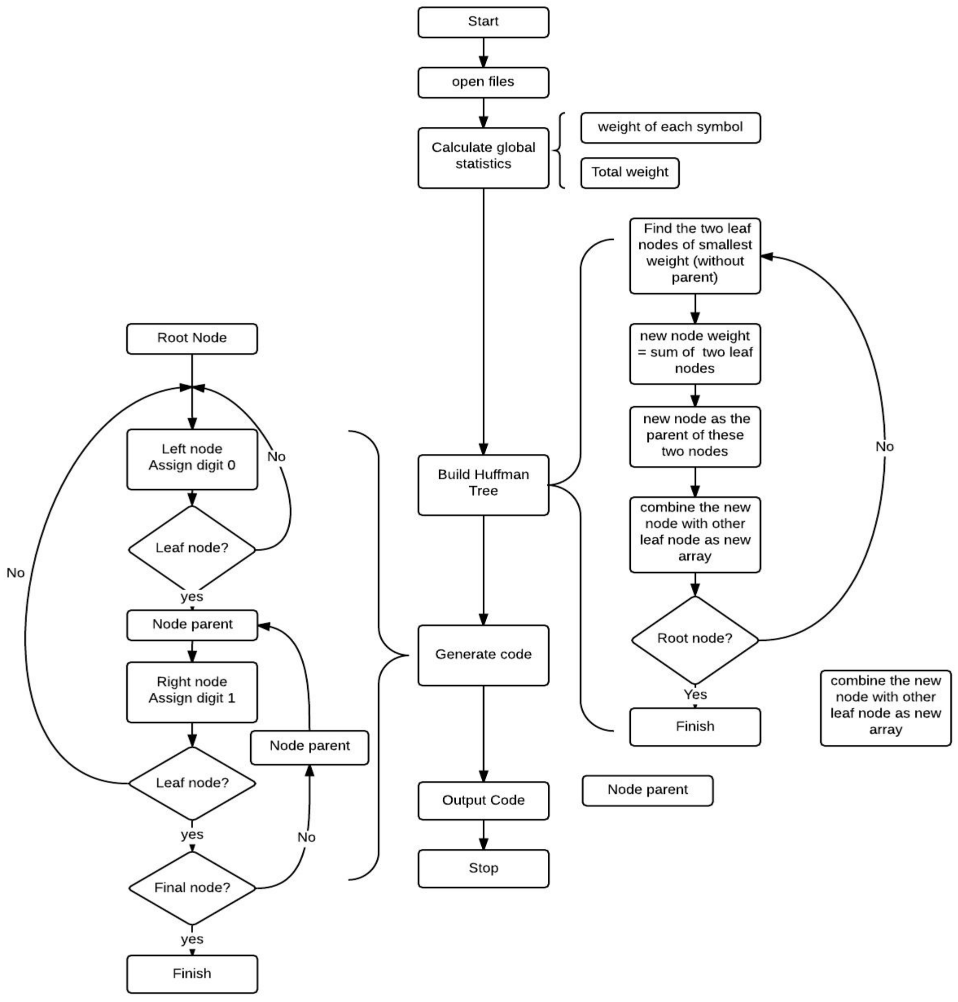

Start
open files
Calculate global statistics
weight of each symbol
Total weight
Find the two leaf nodes of smallest weight (without parent)
new node weight = sum of two leaf nodes
new node as the parent of these two nodes
combine the new node with other leaf node as new array
Root node?
No
Yes
Finish
combine the new node with other leaf node as new array
Node parent
Build Huffman Tree
Generate code
Output Code
Stop
Root Node
Left node Assign digit 0
No
Leaf node?
yes
Node parent
Right node Assign digit 1
Node parent
Leaf node?
No
yes
Final node?
No
yes
Finish

Figure2.2 Procedure of Huffman Coding

B. Data structure and Key function

(1) The key point is to build the Hffman tree according to given global probability.To clearly represent the node and build the tree , construct the structure of the node and code for each leaf node. Meantime,use vector to record the old and new node.

```
//the structure of Huffman node
class HT_NODE
{
public:
    HT_NODE* left;  //left node in the tree
    HT_NODE* right; //right node in the tree
    HT_NODE* parent; //parent node in the tree

    int name;  //node name (symbol)
    double weight; //node weight
    int order; //node order

    HT_NODE() { left = right = parent = NULL; name = 256; weight = 0; order =
            0; };
    HT_NODE(HT_NODE* l, HT_NODE* r, HT_NODE* p, unsigned char n, double w, int
        o)
    {
        left = l;   right = r;  parent = p; name = n; weight = w; order = o;
    }
    ~HT_NODE() { delete left; delete right; delete parent; }
};
typedef vector<HT_NODE*> TreeVector;
```

(2) The **Build_tree()** function is the core function,in which the tree is built through recursive process.

```
/*Generate the Huffman tree*///len is the length of huffman node vector
void Build_tree(double len)
```

2.2.3 Adaptive Huffman Coding

A. Concept and procedure

In real communication process, especially real time decode, we can not get the global statistic . Thus ,Adaptive Huffman coding is developed based on Huffman coding. It is a one pass algorithm, which does not need the global statistics of the input data. It can generate the Huffman codes on the fly.

The procedure flowchart of the algorithm is is shown in Figure 2.3.The key point is to build the tree according to the sibling property. The nodes of the tree are divided into different levels by their weights. The nodes with larger weights are in higher level than the ones with smaller weights. The nodes in the same level are sorted from left to right in an increasing order.[iii] Each input should be told whether be the symbol of exist tree, if not add the new node to the tree and then update tree.Otherwise, add the weight and update the tree.

For the new node ,using NYT(Not Yet Transmitted) to represent the incoming new symbols whose weight is zero. Update tree is based on the sibling property, first make sure it is not the root node ,the check its order in the weight class, if not the max order, swap it with the max order,then go up to its parents and redo the process. Each time when new symbol in ,just code is as original ,then update the tree, if exist, generate as huffman code , after which update the tree.

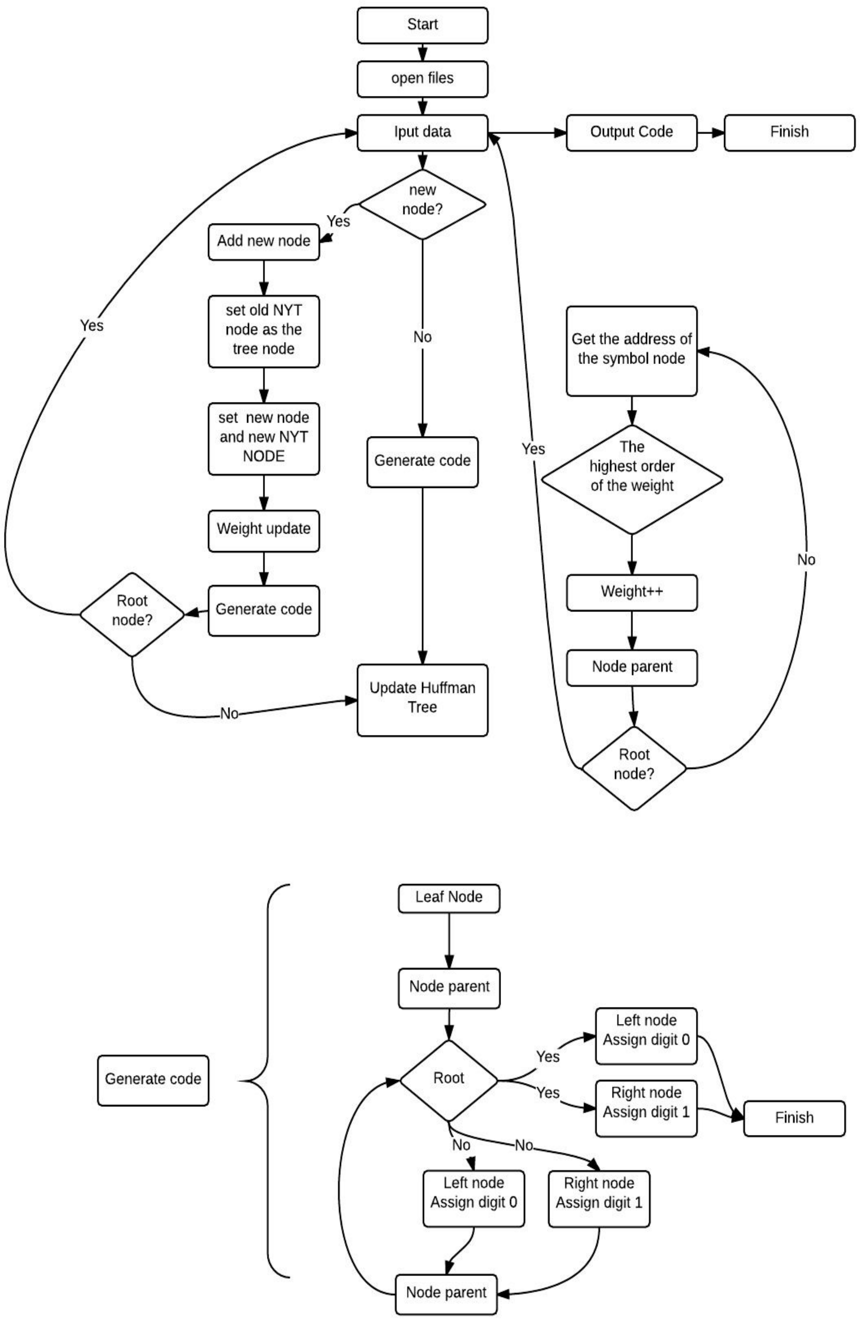


Figure2.3 Procedure of Adaptive Huffman Coding

B. Data structure and Key function

(1) using the same structure as the huffman tree. But using Tree_node[512] to represent each node and NY_NODE to represent the NYT node.

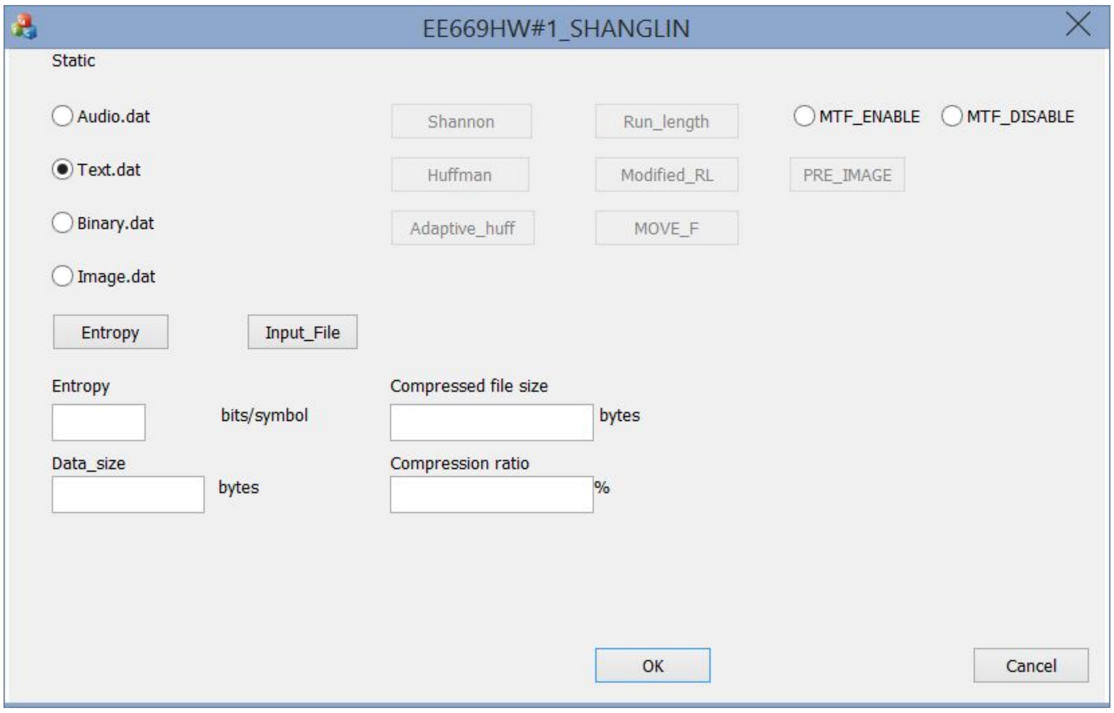
```
HT_NODE *NY_NODE = new HT_NODE(NULL, NULL, NULL, 0, 0, 512);// The NYT node
```

(2)According the flow char,there are three core function , **Add_Root()** aims to the new root,including generate new leaf node and NYT node. **Swap_node()** and **Update_tree()** is key function to update the tree though recursive process. **AD_generate_code()** is to generate the code.

```
void Add_Root(unsigned symbol)
void Swap_node(HT_NODE *Na_parent, HT_NODE *Nb_parent, HT_NODE *Node_a, HT_NODE
*Node_b)
void Update_tree(HT_NODE *NODE_root)
void AD_generate_code(HT_NODE *root, int name,Sample_Code&scode)
```

2.3 Results

The program panel and statistic result are shown below:



EE669HW#1_SHANGLIN

Static

Audio.dat

[✓ Text.dat]

Binary.dat

Image.dat

MTF_ENABLE MTF_DISABLE

Entropy: bits/symbol

Data_size: bytes

Compressed file size: bytes

Compression ratio: %

Figure2.4 The Application UI panel

*The equation for compression ratio calculation is:

$$\text{ratio} = \frac{compressed\_file\_size}{original\_file\_size} \times 100\% \qquad (2)$$

Shannon-Fano Coding:

	Entropy *(bits/symbol)*	Bit Average *(bits/symbol)*	Compression Redundanc *(bits/symbol)*	Original File Size *(bytes)*	Compressed File Size *(bytes)*	Compression Ratio (%)
Audio	6.455944	6.513183594	0.057240	65536	53356	81.41
Binary	0.141036	1	0.858964	262144	32768	12.50
Text	4.439412	4.477426766	0.038015	8705	4872	55.97
Image	7.518736	7.568023682	0.049288	262144	247989	94.60

Table 1 Shanno-Fano coding compression Result

Huffman Coding:

	Entropy *(bits/symbol)*	Bit Average *(bits/symbol)*	Compression Redundanc *(bits/symbol)*	Original File Size *(bytes)*	Compressed File Size *(bytes)*	Compression Ratio (%)
Audio	6.455944	6.489990234	0.034046	65536	53166	81.12
Binary	0.141036	1	0.858964	262144	32768	12.50
Text	4.439412	4.476507754	0.037096	8705	4871	55.96
Image	7.518736	7.547546387	0.028810	262144	247318	94.34

Table 2　Huffman coding compression Result

Adaptive Huffman Coding

	Entropy *(bits/symbol)*	Bit Average *(bits/symbol)*	Compression Redundanc *(bits/symbol)*	Original File Size *(bytes)*	Compressed File Size *(bytes)*	Compression Ratio (%)
Audio	6.455944	6.789672852	0.333729	65536	55621	84.87
Binary	0.141036	1.020019531	0.878984	262144	33424	12.75
Text	4.439412	5.054566341	0.615154	8705	5500	63.18
Image	7.518736	7.629699707	0.110964	262144	250010	95.37

Table 3　Adaptive Huffman coding compression Result

2.4 Discussion

2.4.1 Analysis of each algorithm individually

(1) Shanno-Fano

For shannon_fano coding, we can get several conclusion from above tables :

A. Four kinds of files all do not reach their theoretical bounds and bit averages are larger than the entropy;

B. The compression ratio of four files are different. The compression ratio depends on the input data.

It achieves best compression ratio in binary data file, 12.50%. The reason for compression is that the symbol 0 is encoded as 0 and the symbol 1 is encoded as 1,while we use 1 bit to store one symbol 0 or 1 while the original file uses 8 bits(1 byte). So the compressed file size is 1/8 of the original file size.

The algorithm gets bad result in image and audio data files,and worst result in image data file, 94.60%.

The algorithm gets qualified result in Text file, 55.97%.

(2) Huffman

For Huffman coding, we can get several conclusion from above tables :

A.Four kinds of files all do not reach their theoretical bounds and bit averages are larger than the entropy;

B.The compression ratio of four files are different. The compression ratio depends on the input data.

It achieves best compression ratio in binary data file, 12.50%. The reason for compression is that the symbol 0 is encoded as 0 and the symbol 1 is encoded as 1,while we use 1 bit to store one symbol 0 or 1 while the original file uses 8 bits(1 byte). So the compressed file size is 1/8 of the original file size.

The algorithm gets bad result in image and audio data files,and worst result in image data file, 94.34%.

The algorithm gets qualified result in Text file, 55.96%.

(3) Adaptive Huffman

For Adaptive Huffman coding, we can get several conclusion from above tables :

A.Four kinds of files all do not reach their theoretical bounds and bit averages are larger than the entropy;

B.The compression ratio of four files are different. The compression ratio depends on the input data.

It achieves best compression ratio in binary data file, 12.75%. The reason for compression is that the symbol 0 is encoded as 0 and the symbol 1 is encoded as 1,while we use 1 bit to store one symbol 0 or 1 while the original file uses 8 bits(1 byte). But consider it may first use 2 bits sometimes when the tree achieve the state below,thus its compression ratio is slightly larger than the 12.5%;

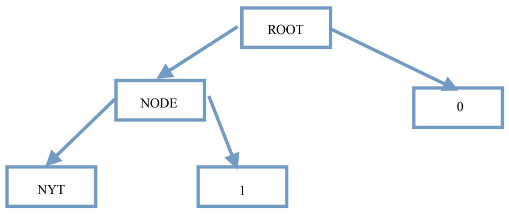


Figure2.5 The Two bits for the AD Huffman

The algorithm gets bad result in image and audio data files,and worst result in image data file, 95.37%.

The algorithm gets qualified result in Text file, 63.18%.

2.4.2 Comparison of different algorithm

(1) Shanno-Fano and Huffman coding

The Huffman coding algorithm is slightly better than the Shanno-Fano coding except the binary data file. And for image and audio files, the optimal effect of Huffman are more obvious.

(2) Huffman and adaptive Huffman coding

The compression ratios of adaptive Huffman coding are almost equal to the Huffman coding and little worsen. Just like the binary files, adaptive huffman coding sometimes uses longer code than the Huffman coding.

2.4.3 Final conclusion and discussion

(1) All algorithm do not reach their theoretical bounds and have different compression ratio for different files , get the best for the binary and the worst for the image;

(2) Shanno-Fano coding is a root-to-leaf method while Huffman code is a leaf-to-root method. In other word , Huffman is the optimal method of the shannon .Huffman coding could improve the performance compared to the shannon,but not obvious. While the improvement get bigger for the image and audio files. In general, Shanno-Fano coding may could get the same result as Huffman coding in best cases, while it can never outperform Huffman coding.

(3) Adaptive huffman have almost same ratio as huffman,even slightly worsen. But the Huffman coding algorithm is a 2 pass procedure which needs the global statistics to generate the Huffman tree. The adaptive Huffman coding is a 1 pass algorithm. Thus, the Adaptive huffman is more practical in real life.

Problem 3: Run-Length Coding

3.1 Abstract and Motivation

Run-length encoding (RLE) is a very simple form of data compression in which runs of data are stored as a single data value and count, rather than as the original run. This is most useful on data while there are strings of many repeated symbol.While in practical experience, the phenomenon is common, for example, simple graphic images such as icons and line drawings. Especially,most time we will also use the pre-processing,after which ,the data become more efficient for run-length coding. Such as DCT and ZigZag scan. Besides, the move to front algorithm also converge the symbol to small value which will also improve the performance of compression procedure.

3.2 Approach and Procedures

3.2.1 Basic Scheme

A. Concept and procedure

Each symbol is represented by name and count.Thus the max count each time is 255,we will cut it into several part for count>255. The encoder procedure flowchart of the Run-length basic scheme is shown in Figure 3.1. For decoder ,each time receive 16 bits,the first 8 bits is the count while the last 8 bits is the symbol.

B. Data structure

Use the structure **RL_NODE** to include the name and count. Then record data the the vector **RLCode_arr**.

```
//the structure of Run_length root node
class RL_NODE
{
public:
    unsigned char lenth; //the code length
    unsigned char code; //the code name

    RL_NODE() { lenth = 0; code = 0; };
    RL_NODE(unsigned char l, unsigned char c) { lenth = l; code = c; }
};

typedef vector<RL_NODE*> CODEVector;
CODEVector RLCode_arr;
```

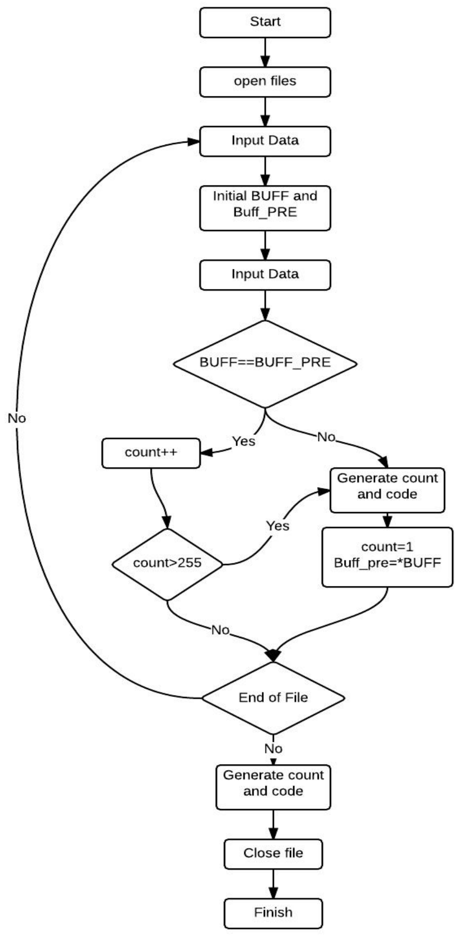


Figure3.1 Procedure of Basic Scheme

3.2.2 Modified Scheme

A. Concept and procedure

Data with MSR is used as count code. For data without MSR and repeat,use data count added with MSR as the code of count. For date shown only once,if it does not have MSR,just record the code name, if it has MSR='1',use count 1 added MSR ('81') before the code. Thus the max count each time is 127,we will cut it into several part for count>127. Each time,push the data into the array .The encoder procedure flowchart of the Run-length basic scheme is shown in Figure 3.2.

For decoder ,each time receive 8 bits, identify whether its MST='1',if so, take it as the count +128 and receive another 8 bits as the code.If not ,just take it as the code.

B. Data structure

Record data including count and code accordingly into the the vector **MLCode_arr.**

```
typedef vector<unsigned char> MLCODEVector;
MLCODEVector MLCode_arr;
```

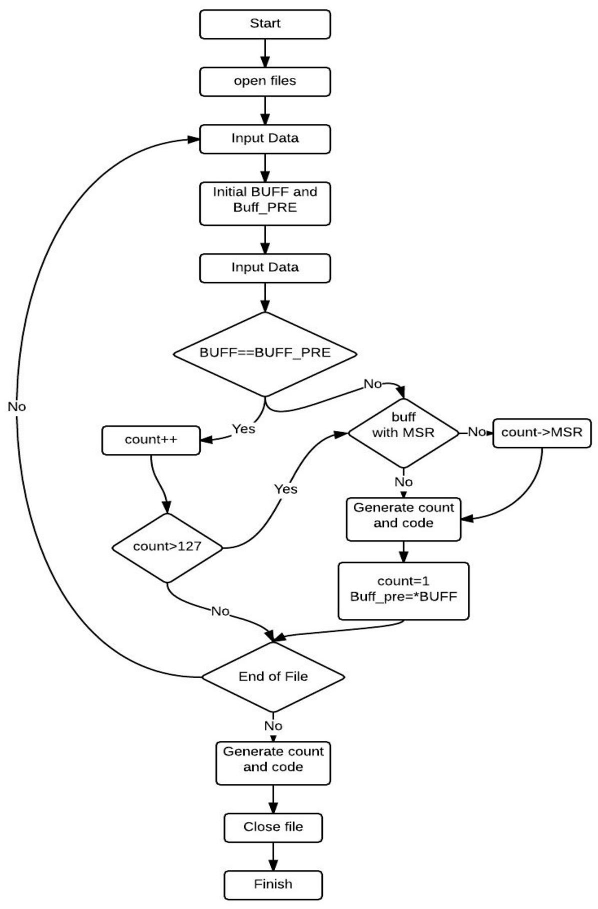


Figure3.2 The Procedure of Modified Run-length

3.2.3 Move-to-Front Transform

A. Concept and procedure

Move-to-Front (MTF) is another excellent pre-processing algorithm that reduces the redundancy in repeat patterns. The MTF scheme transforms the symbol stream into a sequence of index according to an adaptive symbol tables. It simply "moves to front" the last symbol seen such that its index in the table of codes becomes zero. The MTF, as a pre-process stage, could improve the efficiency of other compression method.The encoder procedure flowchart of the Move-to-Front along with Adaptive Huffman is shown in Figure 3.3.

For decoder, first initial the code table,every time receive data take it as index as output the code.Then update the code table as the encoder process.

B. Data structure

Use the structure **CODE_TABLE** to represent the code table the index and code. Then record data the the vector **MvCode_arr**.

```
class CODE_TABLE
{
public:
    int index; //code index
    int code;

    CODE_TABLE() { index = 0; code = 0; };
    CODE_TABLE(unsigned char l, unsigned char c) { index = l; code = c; }
}code_table[256];

typedef vector<unsigned char> MVCOVector;
    MVCOVector MvCode_arr;
```

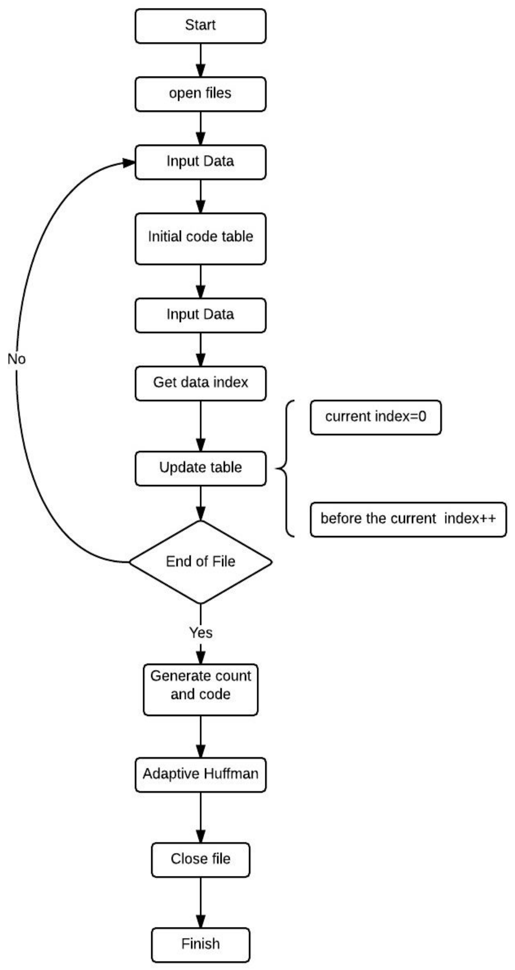


Figure3.3 The Procedure of MTF Huffman

3.3 Results

The statistic result are shown below:

Basic Scheme

	Entropy *(bits/symbol)*	Bit Average *(bits/symbol)*	Compression Redundanc *(bits/symbol)*	Original File Size *(bytes)*	Compressed File Size *(bytes)*	Compression Ratio (%)
Audio	6.455944	13.2487793	6.792835	65536	108534	165.61
Binary	0.141036	0.078430176	-0.062606	262144	2570	0.98
Text	4.439412	15.63790925	11.198497	8705	17016	195.47
Image	7.518736	14.33148193	6.812746	262144	469614	179.14

Table 4 Basic Scheme coding compression Result

Modified Scheme(without pre-processing)

	Entropy *(bits/symbol)*	Bit Average *(bits/symbol)*	Compression Redundanc *(bits/symbol)*	Original File Size *(bytes)*	Compressed File Size *(bytes)*	Compression Ratio (%)
Audio	6.455944	10.40100098	3.945057	65536	85205	130.01
Binary	0.141036	0.141479492	0.000443	262144	4636	1.77
Text	4.439412	8.005514072	3.566102	8705	8711	100.07
Image	7.518736	10.92230225	3.403566	262144	357902	136.53

Table 5 Modified Run-length Scheme compression Result

Modified Scheme(with (MTF)pre-processing)

	Entropy *(bits/symbol)*	Bit Average *(bits/symbol)*	Compression Redundanc *(bits/symbol)*	Original File Size *(bytes)*	Compressed File Size *(bytes)*	Compression Ratio (%)
Audio	6.455944	8.036254883	1.580311	65536	65833	100.45
Binary	0.141036	0.15411377	0.013078	262144	5050	1.93
Text	4.439412	7.943021252	3.503609	8705	8643	99.29
Image	7.518736	9.398254395	1.879518	262144	307962	117.48

Table 6 Modified Run-length Scheme with MTF compression Result

Adaptive Huffman based on MTF

	Entropy *(bits/symbol)*	Bit Average *(bits/symbol)*	Compression Redundanc *(bits/symbol)*	Original File Size *(bytes)*	Compressed File Size *(bytes)*	Compression Ratio (%)
Audio	6.455944	6.446166992	−0.009777	65536	52807	80.58
Binary	0.141036	1.00177002	0.860734	262144	32826	12.52
Text	4.439412	5.81183228	1.372420	8705	6324	72.65
Image	7.518736	7.067409515	−0.451326	262144	231584.875	88.34

Table 7 Adaptive Huffman based on MTF compression Result

3.4 Discussion

3.4.1 Analysis of each algorithm individually

(1) Basic Scheme

From the table above,we can see that the run length can not compress the file.The size of compressed files are larger than the original files for audio,image and text.Only the binary file is compressed.

The run length is only qualified for the data with lots of repeated data.For it generates the another count code for each data. So for text files which does not have much repeated sample ,the compressed files is almost twice size than the original file, the ratio of compression is 195.47%.However, for binary case, in which there are lots of repeated '0' and '255',the performance of run-length is great. Meantime consider it use 1bit rather 8 bit to represent the data , the ratio of compression is 0.98%.

(2) Modified Scheme

From the table above,we can see that the Modified run length Scheme without pre-processing also can not compress the files for audio,image and text. The algorithm have good performance on binary file.

(3) Adaptive Huffman based on MTF

A.Four kinds of files all do not reach their theoretical bounds and bit averages are larger than the entropy;

B.The compression ratio of four files are different. The compression ratio depends on the input data.

3.4.2 Comparison of different algorithm

(1) Basic Scheme and Modified Scheme

Compare the compression ratio between the Basic Scheme and Modified Scheme and get the plot as the Figure 3.4.

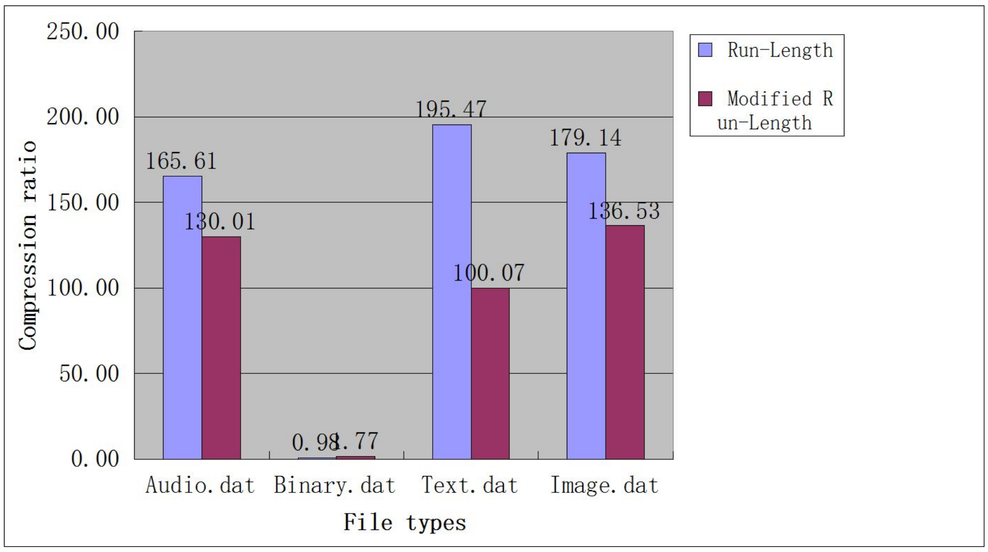


Figure 3.4 Comparison between Basic Scheme and Modified Scheme

From the plot above,we can see the Modified Scheme have better on audio,image and text files.The reason for that is the Modified Scheme avoid the count code for sample appear only once whose MSB is not 1,so it can save code for these symbol compared to the basic Run-length. But it have worsen performance on the binary ,which is the result that the count max is 127 rather than 255 to make MSR=1.

(2) Modified Scheme pre_processing

The result of the Modified Scheme still show it can only compressed the file with many repeated samples . So we consider using the pre-processing to improve the performance. The comparison of compression ratio between using pre-processing and not is shown below.

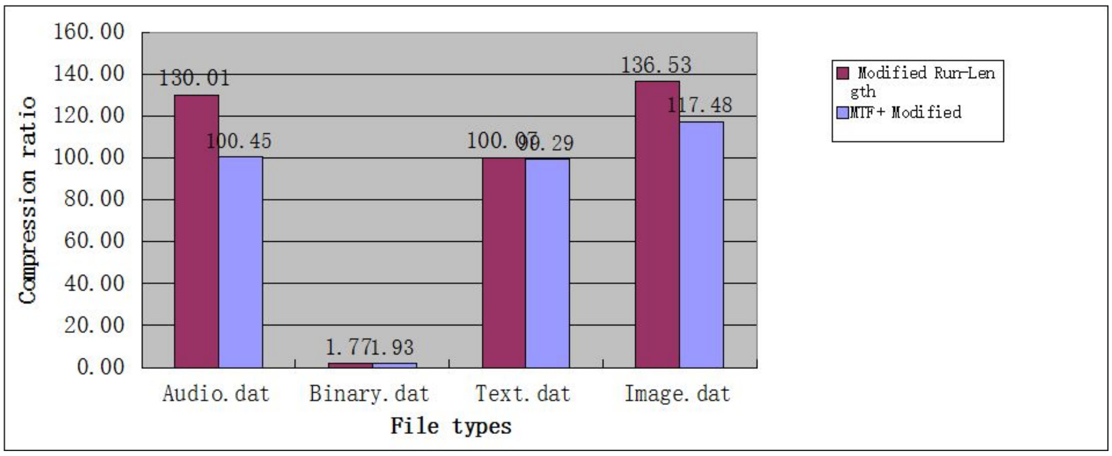


Figure 3.5 Comparison between MTF pre-processing and not

From the plot above,we can see the MTF improve the performance for the result. The reason for that is the MTF can map many larger data to little value compared to original data ,thus improve the chance and number of repeated symbol.

We can also use Zigzag as pre-processing for image.the result is also shown below

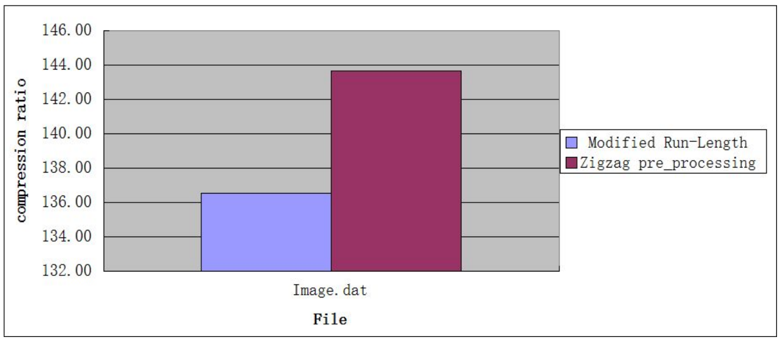


Figure 3.6 Comparison between zigzag pre-processing and not

But the result is not satisfied,for the image pixel data is not strong correlated in the local region. In practice,we can first use DCT,then we can zigzag.

(3) Adaptive Huffman using MTF

We can compare the result from the Adaptive huffman ,the result is shown in the figure below.

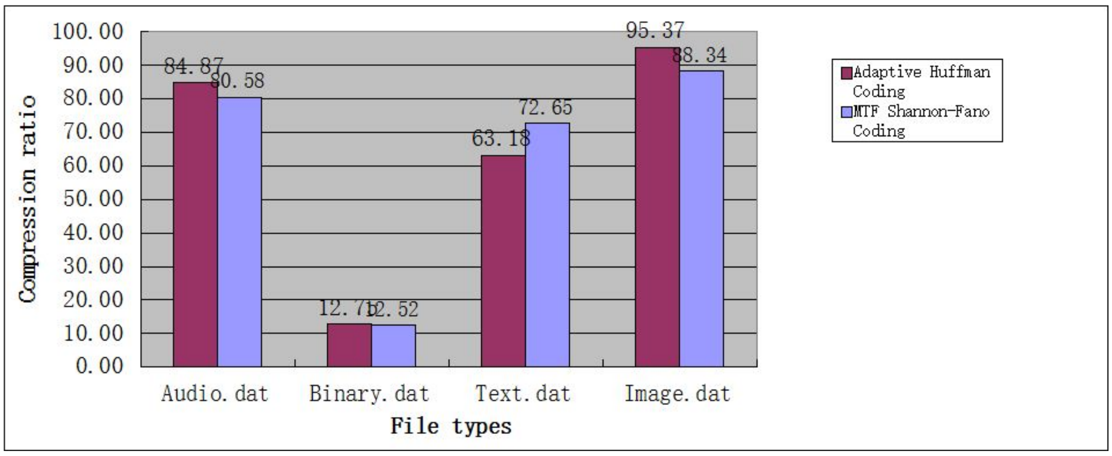


Figure 3.7 Comparison of Adaptive Huffman between using MTF and not

We can find the MTF pre-processing can improve the performance of the audio and image files,but not on the Text data. MTF can improve the performance when the data size is big enough. In general, after the MTF, more samples with larger value could be mapped to the ones with smaller value . The performance is ,on the one hand ,based on the distribution of the sample value , if the data has more repeated chance and uniform distribution ,the performance is better ; on the other hand, if the size is larger enough, the convergence to the little value can have bigger influence on the coding process, thus the improvement of the MTF pre-processing is more obvious.

2.4.3 Final conclusion and discussion

(1) Run-length not always compress the data , it mainly based on the input data. It has good performance when there are enough number of repeated data;

(2) Modified Run-length can improve the performance if Run-length for it may avoid the the

count code for sample appear only once whose MSB is not 1;

(3) Pre-processing can improve the chance and number and repeated samples which can also improve the performance of run-length coding.Meantime,the performance of pre-processing also largely based on the input data;

(4)MTF map more samples with larger value to the ones with smaller value,increasing the probability skew of the sample which can improve the performance of compression ,such as Huffman.

---

i Shannon, C.E. (July 1948). "A Mathematical Theory of Communication". Bell System Technical Journal 27: 379–423.

ii Huffman, David A. "A method for the construction of minimum redundancy codes." *Proceedings of the IRE* 40.9 (1952): 1098-1101.

iii Cormack, Gordon V., and R. Nigel Horspool. "Algorithms for adaptive Huffman codes." *Information Processing Letters* 18.3 (1984): 159-165.