

EE669 Multimedia Data Compression

Homework #1 Report

Part A Huffman Coding

1. *Coding with Global Statistics*

- Motivation and Approach:

General idea about Huffman coding with global statistics is to scan through the input file, get the frequency of the occurrence of every symbol. Then build a Huffman tree according to the frequency of each symbol. Use Huffman tree to build a code table for each symbol.

For compression, the code will first counts the frequency of occurrence of every byte in the input file and put them in corresponding places in the array by using function `count_bytes()`; then it will scale the counts down to fit in an unsigned char by using function `scale_counts()`; after that, it will store the symbol counts in the compressed file by using function `output_counts()`; next step is to build the Huffman tree by using function `build_tree()`; then code each symbol according to the Huffman tree by using function `convert_tree_to_node()`; check to see if there is a command request for debugging information; the last step is to code each symbol from the input file to the compressed file by using function `compress_data()`.

All these steps are implemented in the function `CompressFile()`.

For decompression, the code will first read in the counts for each symbol by using function `input_counts()`; then it will build a Huffman tree using the same algorithm as compression by using the function `build_tree()`; check to see if there is a command request for debugging information; the last step is to read the compressed file and follow the tree to find the correct symbol by using the function `expand_data()`.

All these steps are implemented in the function `ExpandFile()`.

To compute the Entropy of the symbol set, I modified the debugging information, and put two equations in the function `print_model()`. One is to compute the Entropy of the symbol set, the other one is to compute the actual average code length of the symbol set.

To compute the Entropy, first count the total bytes in the input file; then compute the probability of each symbol; the last step is to compute the Entropy.

To compute the Actual average code length, use the `code_bits` and the probability of each symbol.

For detail information, please refer to the source code. Comments are added to each function and some important loops.

- Results:

After compiling, use huff_compress.o to implement compression.

****text.dat File****

The Entropy of the symbol set is: 3.054650

The Actual Average Code Length is: 4.433935

...

Input bytes: 8358

Output bytes: 4719

Compression ratio: 44%

****image.dat file****

The Entropy of the symbol set is: 5.262319

The Actual Average Code Length is: 7.620056

.....

Input bytes: 65536

Output bytes: 62672

Compression ratio: 5%

****audio.dat file****

The Entropy of the symbol set is: 4.481814

The Actual Average Code Length is: 6.500107

.....

Input bytes: 65536

Output bytes: 53463

Compression ratio: 19%

****binary.dat file****

The Entropy of the symbol set is: 0.123508

The Actual Average Code Length is: 1.026820

.....

Input bytes: 65536

Output bytes: 8427

Compression ratio: 88%

After compiling, use huff_decompress.o to implement decompression.

****text.dat file****

Decompressing text1.dat to text2.dat

Using static order 0 model with Huffman coding

...

Original file size: 8358

Decompressed file size: 8358

**** image.dat file ****

Original file size: 65536

Decompressed file size: 65536

**** audio.dat file ****

Original file size: 65536
Decompressed file size: 65536

**** binary.dat file ****

Original file size: 65536
Decompressed file size: 65536

- Discussion:

For compression, we can see that Huff-man coding using Basic Scheme is doing a good job for text file and binary file. The worse case is for image file, which only has a compression ratio of 5%. This means for image file, the symbol probability varies a lot.

As we look at the Entropies, none of the actual average code length of the four types of file equals to its Entropy. That means theoretically speaking, the coding algorithm doesn't reach the theoretical bounds.

Also, we can see that, binary file has the smallest Entropy among the four types of file, while image file has the largest Entropy. This calculated value coincides with the compression ratios.

For decompression, as we can see from the result above, all the decompressed files are of the same size of the original files. And I checked for text.dat, the decompressed file has the exactly same context of the original file. Therefore, the decompressing code works.

2. *Coding with Locally Adaptive Statistics*

- Motivation and Approach:

General idea about Adaptive Huffman coding is to encode the symbols on the fly without the knowledge of global statistics of the symbols. Therefore when encoding, the Huffman tree has to keep updated whenever a new symbol is read in. When decoding, the decoder has to use the same algorithm to build and update the Huffman tree.

For this assignment, I modified the source code from the book “The Data Compression Book” written by Mark Nelson and Jean-loup Gailly.

For compression, it use the function CompressFile(). The code first initialize a Huffman tree with only two symbols, ESCAPE symbol and END_OF_STREAM symbol. Then it goes into a loop to build and update the Huffman tree. Whenever read in a symbol, first check to see if it is in the tree, then encode the symbol. This is done by the function EncodeSymbol(). It takes a symbol, walks from the leaf to the root of the Huffman tree, then gets a reverse order code, convert it an integer and send it out as code for that symbol. After encode a symbol, use UpdateModel to update the tree. When it reaches the end of the input file, it comes out of the loop, encodes the END_OF_STREAM symbol.

For decompression, it use the function ExpandFile(). Same as encoding, the code first initialize a Huffman tree with only two symbols, ESCAPE and END_OF_STREAM. Then it goes into a loop, first use the function DecodeSymbol() to decode the current symbol. It starts at the root node, then go down the tree until it reach a leaf. Check to see if it reaches ESCAPE code or not. If it is ESCAPE, the next symbol is going to be the next eight bits, it is not in the tree. Otherwise, we can decode it from the tree. After decoding the current symbol, use the function UpdateModel to update the tree. When it reaches the END_OF_STREAM symbol, jump out of the loop.

For detail information, please refer to the source code. Comments are added to each function and some important loops.

- Results:

After compiling, use adaptive_huff_com.o to implement compression.

****text.dat File****

Compressing text.dat to text9.dat

Using Adaptive Huffman coding, with escape codes

...

Input bytes: 8358

Output bytes: 4692

Compression ratio: 44%

****image.dat file****

Compressing image.dat to image9.dat

Using Adaptive Huffman coding, with escape codes

.....

Input bytes: 65536

Output bytes: 62609

Compression ratio: 5%

****audio.dat file****

Compressing audio.dat to audio9.dat

Using Adaptive Huffman coding, with escape codes

.....

Input bytes: 65536

Output bytes: 53484

Compression ratio: 19%

****binary.dat file****

Compressing binary.dat to binary9.dat

Using Adaptive Huffman coding, with escape codes

...

Input bytes: 65536

Output bytes: 8423

Compression ratio: 88%

After compiling, use `adaptive_huff_decom.o` to implement decompression.

****text.dat file****

Decompressing `text9.dat` to `text10.dat`

Using Adaptive Huffman coding, with escape codes

Original file size: 8358

Decompressed file size: 8358

**** image.dat file ****

Original file size: 65536

Decompressed file size: 65536

**** audio.dat file ****

Original file size: 65536

Decompressed file size: 65536

**** binary.dat file ****

Original file size: 65536

Decompressed file size: 65536

- Discussion:

For compression, as we can see from the results above, for the four types of data provided, Adaptive Huffman coding scheme gives the same compression ratios as Huffman coding with global statistics. Binary file is compressed the most with a compression ratio of 88%, while image file is compressed the least with a compression ratio of 5%. However, the sizes of the compressed files are not the same between these two schemes. The Adaptive Huffman coding scheme gives a slightly better compression result. The reason why we could not see big difference in these cases is that the symbol sets in these files are not large enough to let adaptive Huffman code to accumulate more statistics. However, judging from the compressed file size, we could still see that Adaptive Huffman coding scheme is better than global statistics scheme.

For decompression, as we can see from the result above, all the decompressed files are of the same size of the original files. And I checked for `text.dat`, the decompressed file has the exactly same context of the original file. Therefore, the decompressing code works.

Part B Lempel-Ziv Coding

- Motivation and Approach:

General idea about LZW coding is to build a dictionary when encoding. Refer the symbol to the dictionary for codes.

For the source code, it uses a 12 bits fixed size code to encode.

For compression, it use the function `CompressFile()` to encode. First it initializes an empty dictionary. Then it goes into a loop. In this loop, the program in new

symbols one at a time from the input file. Then checks to see if the combination of the current symbol and the current code are already defined in the dictionary. If they are not, they are added to the dictionary, and it start over with a new one symbol code. If they are, the code for the combination of the code and character becomes our new code. When it reaches EOF, jump out of the loop.

For decompression, it use the function ExpandFile() to decode. First it goes into a while() loop and read in codes, convert the codes to a string of characters. When the encoder encounters a CHAR+STRING+CHAR+STRING+CHAR sequence, it outputs a code that is not presently defined in the dictionary. This is handled as an exception.

For detail information, please refer to the source code. Comments are added to each function and some important loops.

- Results:

After compiling, use lzw_compress.o to implement compression.

****text.dat File****

Compressing text.dat to text7.dat

Using LZW 12 Bit Encoder

...

Input bytes: 8358

Output bytes: 4721

Compression ratio: 44%

****image.dat file****

Compressing image.dat to image7.dat

Using LZW 12 Bit Encoder

.....

Input bytes: 65536

Output bytes: 77111

Compression ratio: -17%

****audio.dat file****

Compressing audio.dat to audio7.dat

Using LZW 12 Bit Encoder

.....

Input bytes: 65536

Output bytes: 47351

Compression ratio: 28%

****binary.dat file****

Compressing binary.dat to binary7.dat

Using LZW 12 Bit Encoder

.

Input bytes: 65536

Output bytes: 1842
Compression ratio: 98%

After compiling, use lzw_decompress.o to implement decompression.

****text.dat file****

Decompressing text7.dat to text8.dat

Using LZW 12 Bit Encoder

Original file size: 8358

Decompressed file size: 8358

**** image.dat file ****

Original file size: 65536

Decompressed file size: 65536

**** audio.dat file ****

Original file size: 65536

Decompressed file size: 65536

**** binary.dat file ****

Original file size: 65536

Decompressed file size: 65536

- Discussion:

For compression, as we can see from the results above, except for image file, all the other files achieve the goal of compression. Binary file reach a compression ratio of 98%. Since LZW is a dictionary based coding algorithm, the compression ratio depends on the occurrence of symbol sets that are already in the dictionary. Since binary file has a lot of repeated symbols, its compression ratio is very high. However, the image file has very few repeated symbols in the dictionary, therefore, it cannot be nicely compressed.

For decompression, as we can see from the result above, all the decompressed files are of the same size of the original files. And I checked for text.dat, the decompressed file has the exactly same context of the original file. Therefore, the decompressing code works.

Part C Run-length Coding

1. Basic Scheme

- Motivation and Approach:

General idea about the basic scheme is to encode a sequence of repeated symbols as the number of repetition followed by the symbol itself.

For compression, the code use the function CompressFile(). First, it read in an character from the input stream, set the value to variables "charactor" and "charactor_new". This step is initialization. Next, it goes into a loop. In this loop,

it first read in another new character from input file, check to see if it is the same as the old character. If it is the same, count increase one, read the next character. If it is not the same, output count value and the old character. Reset the old character to the new character. Reset count to one.

For decompression, the code use the function ExpandFile(). First, it read in an character as count value. Next, it read in another character as an character code. Then, it output the character code x times according to the count value.

For detail information, please refer to the source code. Comments are added to each function and some important loops.

- Results:

After compiling, use run_length_com.o to implement compression.

****text.dat File****

Compressing text.dat to text3.dat

Using Run-length Coding using Basic Scheme

Input bytes: 8358

Output bytes: 16400

Compression ratio: -96%

****image.dat file****

Compressing image.dat to image3.dat

Using Run-length Coding using Basic Scheme

Input bytes: 65536

Output bytes: 124320

Compression ratio: -89%

****audio.dat file****

Compressing audio.dat to audio3.dat

Using Run-length Coding using Basic Scheme

Input bytes: 65536

Output bytes: 108534

Compression ratio: -65%

****binary.dat file****

Compressing binary.dat to binary3.dat

Using Run-length Coding using Basic Scheme

Input bytes: 65536

Output bytes: 4780

Compression ratio: 93%

After compiling, use run_length_decom.o to implement decompression.

****text.dat file****

Decompressing text3.dat to text4.dat

Using Run-length Coding using Basic Scheme

Original file size: 8358

Decompressed file size: 8358

**** image.dat file ****

Original file size: 65536

Decompressed file size: 65536

**** audio.dat file ****

Original file size: 65536

Decompressed file size: 65536

**** binary.dat file ****

Original file size: 65536

Decompressed file size: 65536

- Discussion:

For compression, as we can see from the result above, only the binary file achieve a nice compression ratio, which is 93%. The other 3 types of file got expanded. This is reasonable, because run-length basic scheme coding method only works in the case when the symbol set has a lot of repeated symbols. From the result, we can see that binary file has a lot of repeated symbols in it, while the other three do not has a lot of repeated symbols. Therefore, the program cannot always compress the input file. The optimal compression ration would be 2 bytes / total bytes.

For decompression, as we can see from the result above, all the decompressed files are of the same size of the original files. And I checked for text.dat, the decompressed file has the exactly same context of the original file. Therefore, the decompressing code works.

2. *Modified Scheme*

- Motivation and Approach:

General idea about the modified scheme is similar to the basic scheme, but when coding, do not encode the count for one singular symbol. For singular symbol which has $MSB = 1$, use "1000 0001" to denote "81" as its count value.

For compression, the code use the function CompressFile(). First, it read in an character from the input stream, set the value to variables "character" and "character_new". This step is initialization. Next, it goes into a loop. In this loop, it first read in another new character from input file, check to see if it is the same as the old character. If it is the same, count increase one, read the next character. If it is not the same and count value is larger than "129" which is "81", output

count value and the old character. Otherwise, only output the old character. Reset the old character to the new character. Reset count to one.

For decompression, the code use the function ExpandFile(). First, it read in an character as count value. Check to see if its MSB is 1 or not. If it is 1, read in another character as an character code. Output it certain times. If its MSB is not 1, output the value as an character.

For detail information, please refer to the source code. Comments are added to each function and some important loops.

- Results:

After compiling, use run_length_mcom.o to implement compression.

****text.dat File****

Compressing text.dat to text5.dat

Using Run-length Coding using Modified Scheme

Input bytes: 8358

Output bytes: 8343

Compression ratio: 1%

****image.dat file****

Compressing image.dat to image5.dat

Using Run-length Coding using Modified Scheme

Input bytes: 65536

Output bytes: 82766

Compression ratio: -26%

****audio.dat file****

Compressing audio.dat to audio5.dat

Using Run-length Coding using Modified Scheme

Input bytes: 65536

Output bytes: 85205

Compression ratio: -30%

****binary.dat file****

Compressing binary.dat to binary5.dat

Using Run-length Coding using Modified Scheme

Input bytes: 65536

Output bytes: 4438

Compression ratio: 94%

After compiling, use run_length_mdecom.o to implement decompression.

****text.dat file****

Decompressing text5.dat to text6.dat
Using Run-length Coding using Modified Scheme
Original file size: 8358
Decompressed file size: 8358

**** image.dat file ****

Original file size: 65536
Decompressed file size: 65536

**** audio.dat file ****

Original file size: 65536
Decompressed file size: 65536

**** binary.dat file ****

Original file size: 65536
Decompressed file size: 65536

- Discussion:

For compression, as we can see from the result above, only text file and binary file have been compressed, with compression ratio of 1% and 94% correspondingly. Image file and audio file were expanded. This means they have very few repeated symbols, and they have a lot of signal symbols with $MSB = 1$. However, the modified scheme is much better than the basic scheme. Especially for symbol sets that do not have a lot of repeated symbols.

For decompression, as we can see from the result above, all the decompressed files are of the same size of the original files. And I checked for text.dat, the decompressed file has the exactly same context of the original file. Therefore, the decompressing code works.

Other ways to improve the compression ratio:

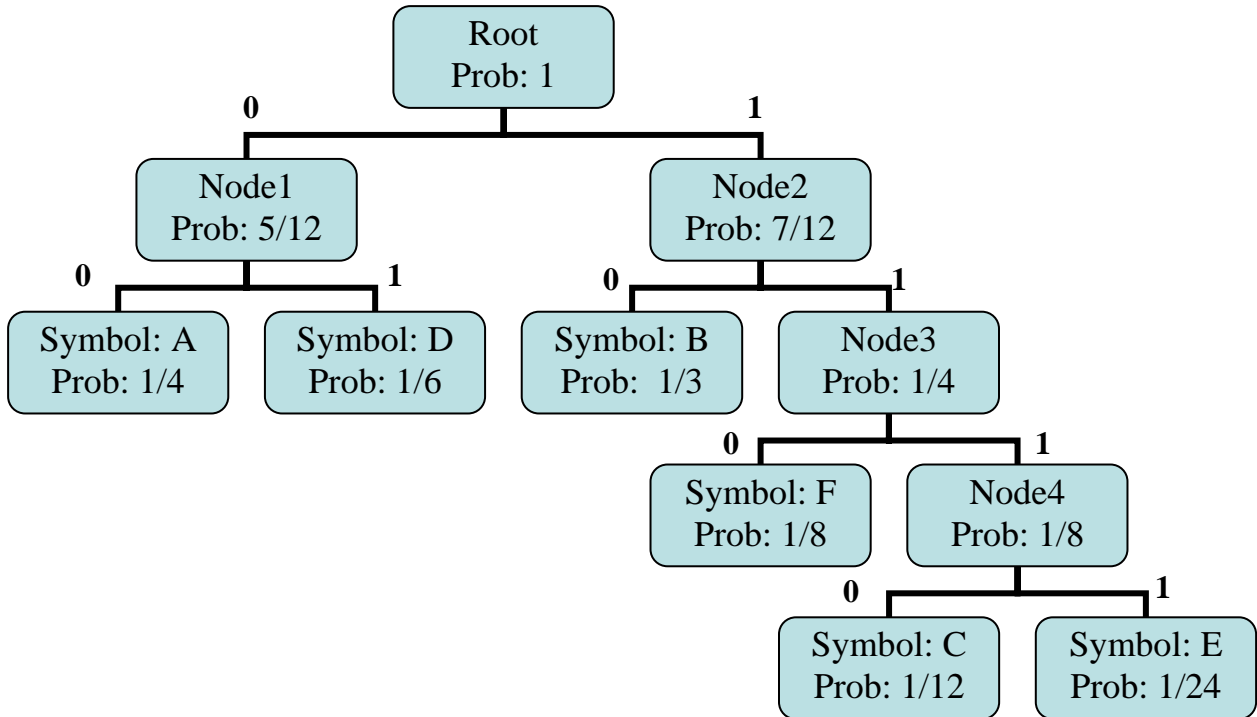
We can use some post-processing scheme to improve the compression ratio. For example, after applying modified run-length coding scheme, we could post-process the compressed the file by using Lempel-Ziv coding scheme.

Written Question

1. Solution:

a)

Huffman Tree:



Code Table:

Symbol	Code
A	00
B	10
C	1110
D	01
E	1111
F	110

b)

Average code word length of this code:

$$\begin{aligned}
 \text{Code Word Length} &= \sum \text{Code_Bits} (S_i) * \text{Probability} (S_i) \\
 &= 2*1/4+2*1/3+4*1/12+2*1/6+4*1/24+3*1/8 \\
 &= 2.375
 \end{aligned}$$

c)

The lower bound for the achievable average codeword length is the Entropy of this symbol set.

$$\begin{aligned}
 \text{Entropy} &= - \sum \text{Probability} (S_i) * \log_2(\text{Probability}(S_i)) \\
 &= 2.3239
 \end{aligned}$$

2. Solution:

The coding table is not a valid Huffman Code table. Because the codes for symbol 'C', 'F' and 'G' is not prefix-free. Huffman Code should be a prefix-free code. It can be used to encode a sequence with the corresponding symbol set, because each symbol has a code. However, we cannot decode it, because the decoder cannot distinguish the codes for 'C', 'F' and 'G'. Therefore, we cannot use this code table.

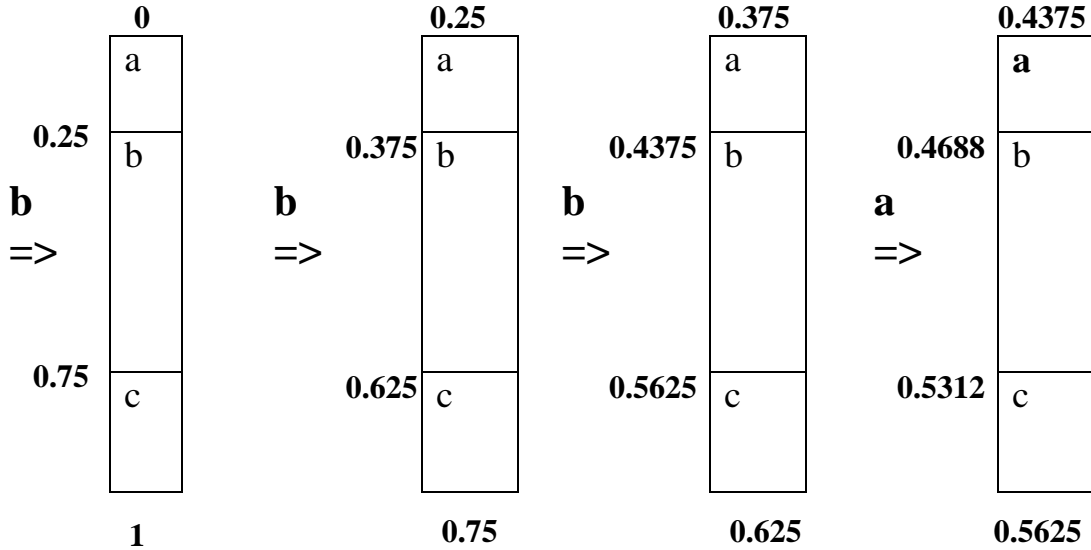
3. Solution:

a)

The probability that 'abbca' occurs is:

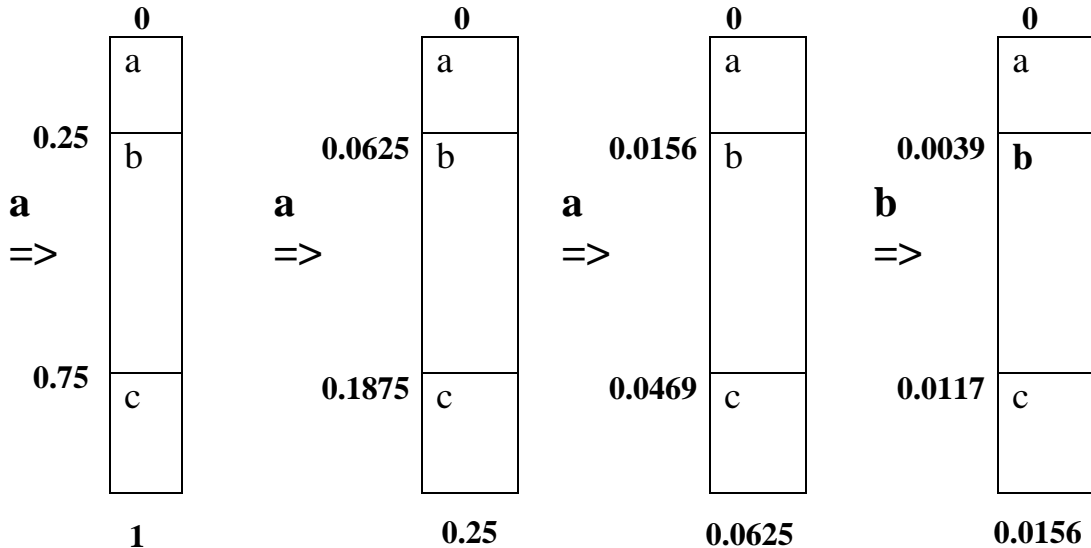
$$P = 0.25 * 0.5 * 0.5 * 0.25 * 0.25 = 0.0039$$

b) Using Arithmetic Coding to code the string 'bbba'.



As we can see from the detailed steps above, 'bbba' lies in the interval [0.4375, 0.5625). Therefore, we can pick 0.4375 which is '0.0111' (binary) to code the string.

c)



'0.0000001' equals 0.0078 in decimal system. As we can see from the plot above, it lies in the interval of [0.0039, 0.0117). Therefore, it can be decoded as 'aaab'.

Additional Information: README.txt in Zip file

EE669 Homework #1
Submission Date: February 2, 2007
Name: Xiao ZHANG
USC ID: 5044493461
Email: zhangxia@usc.edu
Platform: Unix(Aludra)
Editor: Emacs
Compiler: gcc

Files Included in the zip file:

1. huff-entropy.c
(a Huffman Coding Module using Global Statistics coding algorithm, will show entropy)
2. adaptive-huff.c
(an Adaptive Huffman Coding Module modified from the source code in "The Data Compression Book" by Mark Nelson and Jean-loup Gailly)
3. lzw12.c
(a Lempel-Ziv Coding Module)
4. run-length.c
(a Run Length Coding Module using Basic Scheme)
5. run-length-modified.c
(a Run Length Coding Module using Modified Scheme)
6. bitio.h
(bitio.c header file)
7. bitio.c
(a file to implement bit oriented routines)
8. errhand.h
(errhand.c header file)
9. errhand.c
(a file to support error-handling mechanism)
10. main.h
(main-c.c and main-e.c header file)
11. main-c.c
(a compression module)
12. main-e.c
(a decompression module)
13. compile.bat
(a batch file contains compile information)
14. README.txt

Compile Information:

cc -o huff_compress -lm main-c.c bitio.c errhand.c huff-entropy.c

```
cc -o huff_decompress -lm main-e.c bitio.c errhand.c huff-entropy.c
```

```
cc -o adaptive_huff_com main-c.c bitio.c errhand.c adaptive-huff.c  
cc -o adaptive_huff_decom main-e.c bitio.c errhand.c adaptive-huff.c
```

```
cc -o lzw_compress main-c.c bitio.c errhand.c lzw12.c  
cc -o lzw_decompress main-e.c bitio.c errhand.c lzw12.c
```

```
cc -o run_length_com main-c.c bitio.c errhand.c run-length.c  
cc -o run_length_decom main-e.c bitio.c errhand.c run-length.c
```

```
cc -o run_length_mcom main-c.c bitio.c errhand.c run-length-modified.c  
cc -o run_length_mdecom main-e.c bitio.c errhand.c run-length-modified.c
```

Usage

After using the above compile command

1. huff_compress inputfile outputfile [-d]
(Implement Huffman coding with global statistics. using "-d" will show debugging information including the Entropy and the Average Code Length)
huff_decompress inputfile outputfile [-d]
(Implement Huffman decoding scheme. using "-d" will show debugging information including the Entropy and the Average Code Length)
2. adaptive_huff_com inputfile outputfile
(Implement Adaptive Huffman coding)
adaptive_huff_decom inputfile outputfile
(Implement Adaptive Huffman decoding)
3. lzw_compress inputfile outputfile
(Implement Lempel-Ziv Coding)
lzw_decompress inputfile outputfile
(Implement Lempel-Ziv Decoding)
4. run_length_com inputfile outputfile
(Implement Run-length Coding with Basic Scheme)
run_length_decom inputfile outputfile
(Implement Run-length Decoding with Basic Scheme)
5. run_length_mcom inputfile outputfile
(Implement Modified Run-Length Coding)
run_length_mdecom inputfile outputfile
(Implement Modified Run-Length Decoding)

For detail information, please refer to the source code and the Report