Context-Based Arithmetic Encoding

陳永諭, 0816203

I. Introduction

This report compares compression efficiency of 3 different approaches: arithmetic coding, context-based arithmetic coding, and Huffman coding. Below are frequently used notations and their definitions:

- $Acc(x)$: Accumulated count of symbol x, where symbols are sorted in lexicographical order.
- $Acc(x-1)$: Here, "$x-1$" denotes the symbol preceding "x" by 1 in lexicographical order. Therefore, $y=x-1$ implies $Acc(x-1)=Acc(y)$. $Acc(x-1)=0$ if x is the first symbol.
- l: Lower limit of the current interval.
- u: Upper limit of the current interval.
- Len: Word length of l & u in integer implementation.
- ESC: The escape symbol.
- S_{exc}: Symbols to be excluded by Exclusion Principle.
- $E[l(X)]$: Average codeword length (bits) of symbols.
- $H(X)$: Entropy of symbol distribution.

II. Arithmetic Encoding Implementations

A. StaticAccTable

Hold the accumulated count (fixed) of symbols. Given a symbol x, it returns info required for encoding. Specifically, $Acc(x-1)$, $Acc(x)$ and the total number of symbols t.

In addition, the table supports Exclusion Principle. $Acc(x-1)$, $Acc(x)$ and t are calculated without counting symbols in S_{exc} when it is provided.

B. BinaryNum

An integer implementation of l and u such that they can be rescaled and updated conveniently.

C. ArithmeticEncoder

The class keeps track of l, u and the count of E_3 rescales throughout the encoding process. Given info obtained from ***StaticAccTable***, l and u are rescaled (codewords are generated at the same time) and updated according to arithmetic coding algorithm.

D. File Compression

Files can be compressed under 2 modes:

1. Bit mode: treat the file as a bit sequence with 2 different symbols – 0 and 1.
2. Byte mode: treat the file as a byte sequence with 256 different symbols.

Given mode and Len, the source file is compressed by these steps:

1. Calculate symbol distribution.
2. Proportionally shrink the count of each symbol if the total count is too larger for Len.
3. Construct ***StaticAccTable*** with the distribution.
4. For each symbol x in the source file:
   - Obtain info corresponding to x from ***StaticAccTable***.
   - ***ArithmeticEncoder*** encodes x with the info.
   - Write the codewords (if any) to the output file.

III. Context-based Encoding Implementations

A. AdaptiveAccTable

Similar to ***StaticAccTable***, it offers info required for encoding and supports Exclusion Principle. However, the table is not fixed, and it is updated every time a symbol is encoded. There are 3 different implementations for the count of ESC:

1. <u>Method A</u>

   Assign a fixed count of 1.

2. <u>Method C</u>

   Let the number of unique symbols (excluding ESC) in the table be n. The count is set to n if $n>0$. Otherwise, it is set to 1.

3. <u>Method B</u>

   Same as Method C but the count of each symbol is deflated by 1 (except ESC) if its count is greater than 1.

B. NegContextTable

Hold symbol distributions with ***StaticAccTable*** for the -1-order context. Each symbol has a fixed count of 1.

C. ContextTable

Maintain symbol distributions adaptively with ***AdaptiveAccTable*** for non-negative-order contexts.

D. ContextBasedEncoder

This class implements the algorithm for context-based arithmetic encoding. Given a symbol x and its context, x is encoded by these steps:

0. $S_{exc} \leftarrow \emptyset$,

   $C \leftarrow$ the given context

1. Find the ***ContextTable*** T corresponding to C.

```
2. If $T$ exists and $x \in T$:
        Get the info corresponding to $x$ from $T$'s ***AccTable***. ***ArithmeticEncoder*** encodes with the info. Go to **step 4**.
   Else if $T$ exists but $x \notin T$:
        Get the info corresponding to $ESC$ from $T$'s ***AccTable***. ***ArithmeticEncoder*** encodes with the info.
   Else:
        Create a new ***ContextTable*** for $C$.
3. If $C$ is an empty string, encode $x$ with ***NegContextTable***.
   Else, remove the first symbol from $C$ and go to **step 1**.
4. $C \leftarrow$ the given context.
   While true:
        Get the ***ContextTable*** $T$ for $C$.
        Increment the count of $x$ in $T$.
        Break the loop if $C$ is an empty string, and remove the first symbol from $C$ otherwise.
```

To adopt Exclusion Principle:

- Provide S_{exc} to ***ArithmeticEncoder*** when encoding.
- Add all symbols (except ESC) in T to S_{exc} when ESC is encoded (the else if case in **step 2**).

E. File Compression

Files can be compressed under bit or byte mode as in arithmetic encoding. Given compression mode, Len, method for ESC count, and whether the Exclusion Principle applies, each symbol x in the source file is compressed by:

1. Read its context C (as long as available but no longer than 2).
2. ***ContextBasedEncoder*** encodes x with C.
3. Write the codewords (if any) to the output file.

IV. RESULTS

For all experiments in this report, Len is set large enough to accommodate the count of all symbols in the source file. Specifically, it is set to 33 for bit mode and 30 for byte mode.

A. Arithmetic Encoding

The performance can be found in TABLE I. Under bit mode, $H(X)$ is very close to 1, which is exactly the length of symbol. This indicates that the optimal compression ratio is roughly 0. As a result, the file is barely compressed even though $E[l(X)]$ is very close to $H(X)$.

On the other hand, the file is shrunk by 12.98% when treated as byte sequence. The ratio is slightly better than that of Huffman Coding (12.65%, TABLE II).

TABLE I. RESULTS OF ARITHMETIC CODING

Mode	$E[l(X)]$	***H(X)***	***Compression Ratio***
Byte	6.96	6.96	12.98%
Bit	1.00	1.00	0.08%

TABLE II. RESULTS OF HUFFMAN CODING

Bytes per Symbol	$E[l(X)]$	$H(X)$	***Compression Ratio***
1	6.99	6.96	12.65%
2	8.59	8.57	46.30%
4	8.92	8.89	72.11%

B. Context -Based Arithmetic Encoding (Byte Mode)

The results are summarized in TABLE III. Before further discussions, let's define the following:

- n_t: number of symbols encoded by context t
- $H(X_t)$: entropy of the ***AccTable*** of context t
- T_l: the set of all contexts with order l
- N_l: total number of symbols encoded by T_l
- Weighted entropy of T_l: $\sum_{t \in T_l} \frac{n_t}{N_l} \times H(X_t)$

TABLE III. RESULTS OF CONTEXT-BASED CODING (BYTE MODE)

Method	**Exclusion Principle**	$E[l(X)]$	***Compression Ratio***
A	Yes	2.26	71.79%
	No	2.26	71.79%
B	Yes	2.26	71.81%
	No	2.26	71.76%
C	Yes	2.25	71.82%
	No	2.26	71.77%

As order increases, the weighted entropy of contexts drop significantly from 8.00 to 2.25 (TABLE IV). And the resulting compression ratio is around 72%, where Huffman Coding must extend the symbols to 4 bytes to achieve a comparable ratio (72.11%, TABLE II).

As for the parameters, neither Exclusion Principle nor count method of ESC has conspicuous impacts. Probably because there are more than 244 million symbols in the source file yet fewer than 10,000 different contexts (TABLE IV). The great majority of symbols are encoded without escaping. Exclusion Principle and count method of ESC, which are designed for escaping, are rarely used as a result.

TABLE IV. WEIGHTED ENTROPY OF CONTEXTS (BYTE MODE)

Context Order	Number of Contexts	Weighted Entropy
-1	1	8.00
0	1	6.96
1	256	6.11
2	9494	2.25

C. Context -Based Arithmetic Encoding (Bit Mode)

TABLE V shows the symbol distribution of each context. Unfortunately, high order contexts do not effectively skew the distribution away from that of the 0-order context. Consequently, the performance is not improved from arithmetic coding by far. The results are displayed in TABLE VI. Also, Exclusion Principle and count method of ESC have neglectable influence.

TABLE V. SYMBOL DISTRIBUTIONS OF CONTEXTS (BIT MODE)

Context	Symbol Distribution		*Entropy*
	0	1	
‘’	48.14%	51.86%	1.00
‘0’	49.29%	50.71%	1.00
‘1’	47.21%	52.79%	1.00
‘00’	50.37%	49.63%	1.00
‘01’	44.59%	55.41%	0.99
‘10’	48.16%	51.84%	1.00
‘11’	49.92%	50.08%	1.00

TABLE VI. RESULTS OF CONTEXT-BASED CODING (BIT MODE)

Method	Exclusion Principle	$E[l(X)]$	*Compression Ratio*
A	Yes	1.00	0.25%
	No	1.00	0.25%
B	Yes	1.00	0.25%
	No	1.00	0.25%
C	Yes	1.00	0.25%
	No	1.00	0.25%

V. FUTRURE RESEARCH

Arithmetic Coding outperforms Huffman Coding only by a small margin. One possible reason is that the symbol distribution is not highly skewed as shown in Fig.1. It would be interesting to use different files and find out what kinds of distribution favor which of the 2 coding methods.

Moreover, context-based coding with a max order of 2 already performs as good as Huffman Coding with symbols extended to 4 bytes. Trying a higher max order may further enhance the performance.

Finally, Len was set large enough such that distribution never shrinks in $\boldsymbol{AccTable}$. If the distribution varies a lot throughout the source file, however, occasional shrinking can help the encoder adapt to the most recent distribution and ultimately yields higher compression ratio. Therefore, it is worth it to discover the relationship between Len and compression efficiency.

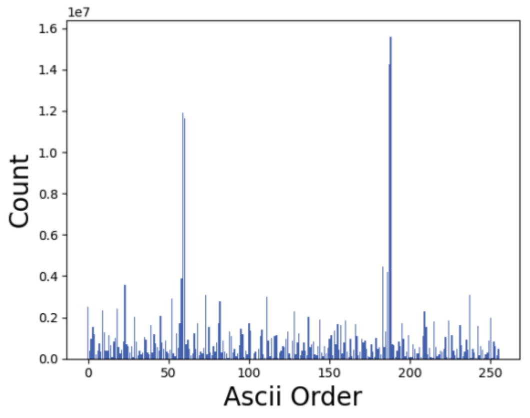

Fig. 1. *Symbol Distribution as byte sequence.*