Huffman-Coding

I. Introduction

This report compares compression efficiency between basic and adaptive Huffman Algorithm. All experiments were conducted on the AlexNet model (in Python pickle format). Finally, an improvement was made on the adaptive algorithm after examining the symbol distribution and experiment outcomes.

Below are frequently used notations in the report and their definitions:

- β: Bytes per symbol.
- $\mathrm{C}(x)$: Codeword of x.
- $l(x)$: Codeword length of x.
- $E[l(X)]$: Average codeword length of source file X.
- ϕ^b: Number of bits used for padding the zipped file.

  If C(source file) has n bits, which is not a multiple of 8, the end of the zipped file must be padded with dummy bits to complete a byte.

  $$\phi^b = 8 - (n \bmod 8)$$

  Else

  $$\phi^b = 0$$

- ϕ^B: Number of bytes used for padding the source file.

  If the source file consists of n bytes, which is not a multiple of β, the end of it must be padded with dummy bytes to complete a symbol.

  $$\phi^B = \beta - (n \bmod \beta)$$

  Else

  $$\phi^B = 0$$

- Σ: The dictionary which indicates optimal codeword length for each symbol appeared in the source file.
- $\mathcal{H}$: Header of the zipped file. It is composed of information required for decoding. For instance, β, ϕ^b, ϕ^B, Σ.

II. Basic Huffman Algorithm

A. Huffman Tree

The tree can be constructed based on either:

1. Codeword Lengths
   In the tree, each symbol is positioned at depth corresponding to the given codeword length. Those with the same length will be sorted lexicographically and aligned leftward in the same level.
2. Symbol Distributions
   The tree first determines the optimal codeword length of each symbol by following the standard Huffman Algorithm. After that, it rebuilds itself with the optimal codeword length as above-mentioned. In this way, the tree is guaranteed to be canonical.

B. Encoder

File is encoded by following steps:

1. Calculate symbol distributions.
2. Build a Huffman Tree with the distributions.
3. Write header $\mathcal{H}$ at the beginning of the zipped file.
4. Encode the contents with the tree and write the corresponding codewords into the zipped file.

C. Decoder

File is decoded by following steps:

1. Parse the header $\mathcal{H}$.
2. Rebuild the Huffman Tree according to Σ.
3. Decode file content with the tree and write the corresponding symbols into the decompressed file.
4. Truncate the decompressed file depending on ϕ^b and ϕ^B.

III. Adaptive Huffman Algorithm

A. Adaptive Huffman Tree

- NYT node

  It keeps track of symbols not yet transmitted. A Codeword of a not yet transmitted symbol is formulated by:

  1. Get the (extended) ascii value of each character.
  2. Transform the value into base 2.
  3. Concatenate each of them.

  For example:

  $NYT(\text{“}ab\text{”})$ = “01100001” + “01100010”
  = “0110000101100010”

- Blocks

  Each block maintains nodes of the same weight with a min heap (by their depths in the tree). The

representative of a block is the one at the top of the heap, which has the smallest depth.

This differs from the standard algorithm where the node with largest node number represents its block. The main reason behind the modification is that updating (when swapping nodes) depths is much easier than node numbers.

- Tree:

  It is constructed and updated according to the standard algorithm, except the selection of block representative when swapping nodes. However, this does not compromise the optimality. Since the chosen representative has equal depth and codeword length compared to the original algorithm.

  A symbol x is encoded by traveling upward from its corresponding leaf to the root (the resulting string should be reversed). The travel starts from the NYT node if it is not yet transmitted and the resulting codeword is concatenated with $NYT(x)$.

  A codeword is decoded by traveling from the root to a leaf node. If the travel ends at the NYT node, the symbol is determined by the $n(=8\beta)$ following bits b^n from the zipped file. Specifically, the symbol will be $NYT^{-1}(b^n)$.

B. Encoder

File is encoded by following steps:

1. Encode the source file with the Adaptive Huffman Tree and write the corresponding codewords into the compressed file.
2. Insert header $\mathcal{H}$ at the beginning of the zipped file.

C. Decoder

File is decoded by following steps:

1. Parse *the header* $\mathcal{H}$.
2. Decode the file content with the Adaptive Huffman Tree and write the corresponding symbols into the decompressed file.
3. Truncate the decompressed file depending on ϕ^b and ϕ^B.

IV. Symbol Distribution

Entropy and number of unique symbols of the file for different $\boldsymbol{\beta}$ is summarized in TABLE I. To further investigate the distribution, the file is divided into chunks of 10 Mb (except the last one). Comparison (by relative entropy) of each chunk to the total distribution is visualized in Fig. 1. Pairwise comparison (by relative entropy) between chunks can be found in Fig. 2.

It is observed that, regardless of $\boldsymbol{\beta}$, the file can be divided into 4 segments as in TABLE II. Total distributions of any 2 segments are quite different. Whereas the distributions of all chunks in the same segment are very similar.

TABLE I. ENTROPY

$\boldsymbol{\beta}$	1	2	4
H(X) (bits)	6.9612	8.5654	8.8935
Unique Symbols	256	2874	7304

TABLE II. File segments

Segment Number	1	2	3	4
Range (Mb)	0~140	140~210	210~220	220~233
Size (%)	60.09	30.04	4.29	5.58

V. Results

Performance of basic and adaptive algorithms are summarized in TABLE III and TABLE IV respectively. It turned out that the results of both algorithms were very close to the theoretical optimum $H(X)$. Despite of larger header size, the basic one slightly outperforms the other. One probable reason is that segment 1 constitutes over 60% of the file. As a result, it has huge impacts on the encoding and decoding of subsequent segments, no matter how the distribution changes.

TABLE III. Performance of Basic Algorithm

β	$E[l(X)]$ **(bits)**	$\frac{E[l(X)]}{H(X)} - 1$	$\frac{E[l(X)]}{\beta}$	***Compression Ratio***
1	6.9881	0.4065%	6.9881	12.6490%
2	8.5907	0.3152%	4.2954	46.3033%
4	8.9163	0.3261%	2.2291	72.1125%

TABLE IV. Performance of adaptive Algorithm

β	$E[l(X)]$ **(bits)**	$\frac{E[l(X)]}{H(X)} - 1$	$\frac{E[l(X)]}{\beta}$	***Compression Ratio***
1	6.9895	0.4065%	6.9881	12.6313%
2	8.5924	0.3152%	4.2954	46.2975%
4	8.9225	0.3261%	2.2291	72.1172%

VI. Improvements on adaptive algorithm

According to the observation above, one key to improvement is undermining the impact of prior distributions. To this end, 2 extra arguments chunk size ($\boldsymbol{K}$) and shrink factor ($\boldsymbol{\alpha}$) are added to the Adaptive Huffman Tree.

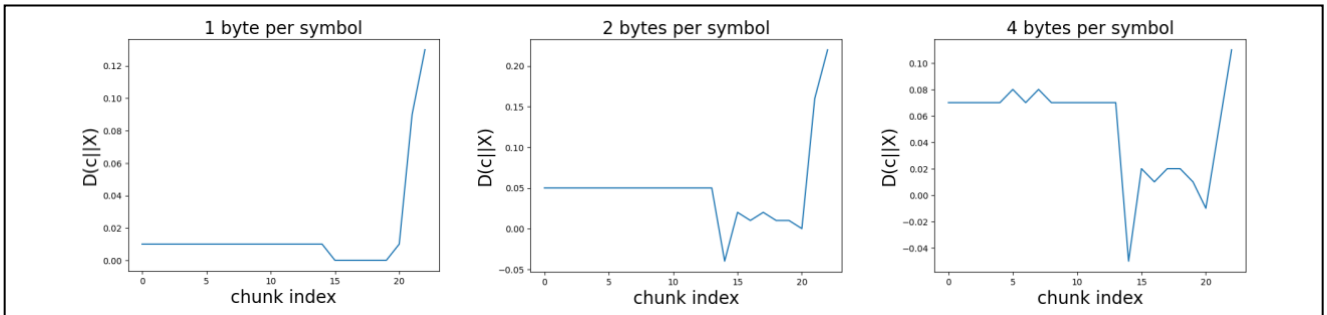

Fig. 2. *Comparison of each chunk to total distribution by relative entropy. Due to high computaional cost, symbols with rare appearence (less than 1 thousand times) were omitted.*

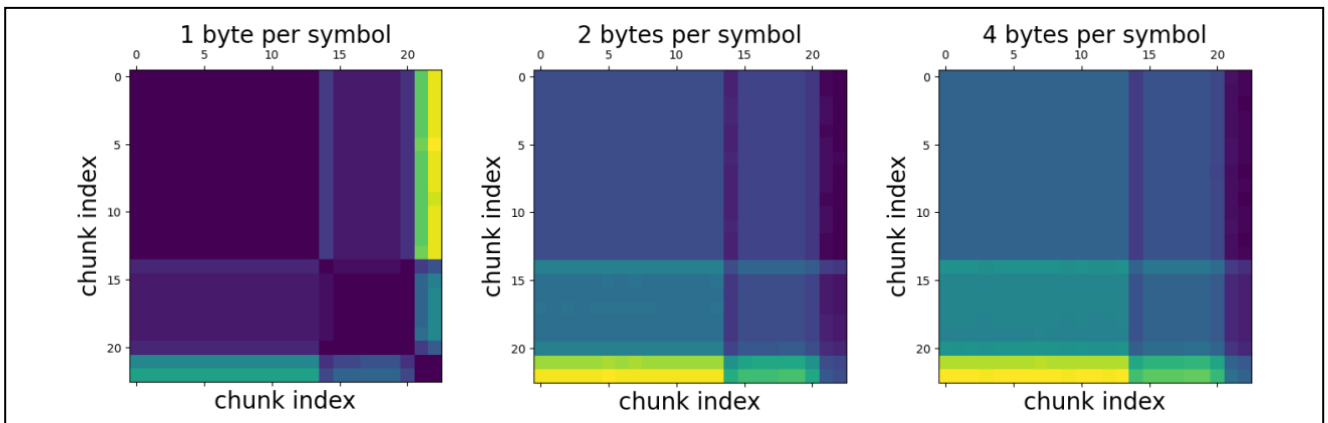

Fig. 1. *Pairwise comparison between chunks by relative entropy. The closer the color is to purple, the smaller the relative entropy. The closer the color is to red, the larger the relatve entropy.*

The source file is divided into chunks of equal size K, except the last one. During encoding and decoding, the tree calls the $shrink$ function every time a chunk is fully processed.

In the $shrink$ function:

1. Weight w of all leaf nodes $= \max(1, \lfloor \frac{w}{\alpha} \rfloor)$.
2. Weights of all internal nodes are updated recursively according to their children.
3. Rebuild the blocks with new weights.

This equivalently divides the occurrence of each symbol by α.

By doing so, the impact of older chunks decays exponentially and the adaptive algorithm is expected to be more "adaptive" toward the most recent chunk. The outcomes of different combinations of arguments are summarized in TABLE V and TABLE VI.

The improved adaptive algorithm not only outperforms the basic algorithm but also $H(X)$. Furthermore, K is more influential to the performance compared to α.

TABLE V. PERFORMANCE OF IMPROVED ALGORITHM ($\beta = 2$)

K \ α	1Mb		4 Mb		10 Mb	
	$E[l(X)]$	$\frac{E[l(X)]}{H(X)} - 1$	$E[l(X)]$	$\frac{E[l(X)]}{H(X)} - 1$	$E[l(X)]$	$\frac{E[l(X)]}{H(X)} - 1$
2	8.5282	-0.92%	8.5368	-0.33%	8.5448	-0.24%
3			8.5368	-0.33%	8.5442	-0.25%
4	8.5282	-0.92%	8.5368	-0.33%		
8	8.5282	-0.92%	8.5368	-0.33%		

TABLE VI. PERFORMANCE OF IMPROVED ALGORITHM ($\beta = 4$)

K \ α	1Mb		4 Mb		10 Mb	
	$E[l(X)]$	$\frac{E[l(X)]}{H(X)} - 1$	$E[l(X)]$	$\frac{E[l(X)]}{H(X)} - 1$	$E[l(X)]$	$\frac{E[l(X)]}{H(X)} - 1$
2	8.8394	-0.61%	8.8480	-0.51%	8.8573	-0.41%
3			8.8480	-0.51%	8.8573	-0.41%
4	8.8394	-0.61%	8.8480	-0.51%	8.8573	-0.41%
8	8.8394	-0.61%	8.8480	-0.51%		

VII. CONCLUSIONS

The basic and adaptive algorithms are comparably efficient in terms of compression ratio. Moreover, the improved

adaptive algorithm can outperform both of above as anticipated. However, the experiments were not conducted comprehensively due to hardware limits. So far, it seems like α plays a neglectable role. The reason behind remains unknown, though. In the future, it will be interesting to try out β other than 2 or 4, different source files, larger α, etc.