Report of Data Compression Project

117010285 Chenhao WU

1 Methodology

I. Huffman Coding Based on Letters

steps to encode

1. Build a Huffman Tree by continuously scanning the text file. Collect a frequency table of each letter;
2. Traverse the Huffman Tree and assign code words to each node

For an sample codebook in `bible.txt`:

character	frequency	codeword
a	f(a)	01110010
b	f(b)	01010101
c	f(c)	01011011
...	...	...

3. Output the compressed text according to the codebook;

II. Huffman Coding Based on Words

steps to encode

1. Transfer the text file to an array of word segments;

   `"In the beginning God created the heaven and the earth..." -> ["In", "\space", "the", ...]`

2. Collect a frequency table of each word;

   For an sample codebook in `bible.txt`:

word	frequency	codeword
\space	f(\space)	00000001110101
A	f(A)	00011000110000
AA	f(AA)	00101010100111
AAINDEX	f(AAINDEX)	00000011101000
ABBEY	f(ABBEY)	01001010111000
ABEDA	f(ABEDA)	01101000100011
...	...	...

3. Output the compressed text according to the codebook;

III. Lempel-Ziv Source Coding

steps to encode

Example: To encode a string `ABBABBBAAAABBABA`

Numerical Position	Subsequence	Codeword
1	A	000 0
2	B	000 1
3	BA	010 0
4	BB	010 1
5	BAA	011 0
6	AA	001 0
7	BBA	100 0

We can obtain a codebook like above by performing Lempel-Ziv Algorithm, and then encode the source text to `0000000101000101011000101000010 0`

steps to decode

Since that the encoding method is algorithmically invertible, the program can perform corresponding inverse operations to decode the encoded message.

IV. Huffman Coding Based on Semantic Statistics

A revised version of Huffman Coding proposed in this project is to analysis the *statistical semantic characteristics* of source text before compression.

As what being discussed in Huffman Coding sections above, the traditional Huffman Coding methods treat the text as a sequence of individual units, which might be letter, word, dual-word, and so on, and assign the prefix code by respective frequency. By using Huffman Coding we can achieve the optimal code since `H(X) <= L(C) <= H(X) + 1` for corresponding distribution. For dual-letter groups in English language, however, if the first letter has been given, with high probability the set we choose the second letter is merely a subset of English Alphabet. For instance, if we have the first letter be *a*, then less-likely the second the letter is also an *a*, so does *k*, *z*,... At the same time, the probability of an *b*, *d*, *s*, *n*... might be higher since we have more words start or end with *ab ad*, *as*, *an*...

Based on this understanding, if the letter before is known, the uncertainty of next letter (entropy) will decrease, and therefore we can achieve a better Huffman Code since the entropy has decreased.

A sample code book of `bible.txt` should look like:

Previous Letter	Consecutive Letter	Codeword
a	\space	11101
	,	111111
	t	110
	l	101
	......	......
b	e	00
	l	101
	o	100
	....	......

The expected code word length is much fewer than the Huffman Coding before. Notice that since the Huffman Tree is individual from letters to letters, which means the codeword of each *previous letter* is prefixed, but the primitive version of codebook does not guarantee the merged code is also prefixed, i.e. `c(a->b) + c(b->c) + ... + c(x_n->x) = c(a->k)` . Hence, before applying the codebook into compression, the program will examine and modify the conflicts of each codeword.

Result

Huffman Coding Based on Letters

File Name	Original Size	Compressed Size (Include Dictionary)	Rate
`alice29.txt`	152.1 KB	43.8 KB	28.80%
`asyoulik.txt`	125.2 KB	37.8 KB	30.19%
`bible.txt`	4.0 MB	1.1 MB	27.5%
`lcet10.txt`	426.8 KB	125.3 KB	29.36%
`plrabn12.txt`	481.9 KB	137.8 KB	28.60%
`world192.txt`	2.5 MB	778.7 KB	30.42%

Huffman Coding Based on Words

File Name	Original Size	Compressed Size + Dictionary Size	Rate
`alice29.txt`	152.1 KB	5.162 KB + 28.411 KB (dict)	22.07%
`asyoulik.txt`	125.2 KB	6.014 KB + 32.26 KB (dict)	30.57%
`bible.txt`	4.0 MB	58.416 KB + 124.60 KB (dict)	4.47%
`lcet10.txt`	426.8 KB	12.256 KB + 65.11 KB (dict)	18.12%
`plrabn12.txt`	481.9 KB	11.369 KB + 98.083 KB (dict)	22.71%
`world192.txt`	2.5 MB	127.94 KB + 194.39 KB (dict)	12.59%

Lempel-Ziv Source Coding

File Name	Original Size	Compressed Size	Rate
`alice29.txt`	152.1 KB	58.3 KB	38.33%
`asyoulik.txt`	125.2 KB	51.1 KB	40.81%
`bible.txt`	4.0 MB	1.3 MB	32.5%
`lcet10.txt`	426.8 KB	151.9 KB	35.59%
`plrabn12.txt`	481.9 KB	177.6 KB	36.85%
`world192.txt`	2.5 MB	868.6 KB	33.92%

Huffman Coding Based on Semantic Statistics

File Name	Original Size	Compressed Size (Include Dictionary)	Rate
`alice29.txt`	152.1 KB	30.2 KB	19.86%
`asyoulik.txt`	125.2 KB	23.8 KB	19.03%
`bible.txt`	4.0 MB	738.8 KB	18.47%
`lcet10.txt`	426.8 KB	89.4 KB	20.95%
`plrabn12.txt`	481.9 KB	95.5 KB	19.82%
`world192.txt`	2.5 MB	552.6 KB	18.25%

Zip Software on Ubuntu

File Name	Original Size	Compressed Size	Rate
`alice29.txt`	152.1 KB	54.6 KB	35.90%
`asyoulik.txt`	125.2 KB	49.1 KB	39.22%
`bible.txt`	4.0 MB	1.2 MB	30.00%
`lcet10.txt`	426.8 KB	145.1 KB	34.00%
`plrabn12.txt`	481.9 KB	195.5 KB	40.57%
`world192.txt`	2.5 MB	695.9 KB	27.18%

Discussion

In this data compression project we have wrote programs for each compression method, and then tested the performance by compressing text files. Included methods are *Huffman Coding on letter*, *Huffman Coding on word*, *LZ Coding*, *Huffman Coding on Semantic Stats*, and *System Compression Tool on Ubuntu*.

As shown above, we found that the performance of Huffman Coding on letter are approximately the same for the English language. Huffman Coding on words works slightly better than the former one, and achieved a significant better performance on text with small words set like `bible.txt` .

For Zip and LZ Coding, since they based on the same compression mechanism, the performance of them are very closed. For files with longer length, the lower compression rates can be achieved.

For Huffman Coding based on Semantic Statistics, it can *achieve an lowest average compression rate among these methods*, ranging from 18.25% - 20.95%. Since the key of codebook is based on letter-level semantics, the dictionary size is also bounded within 5 KB - 10 KB.

Notes

- Programming Language: C++
- Build Method: CMake
- Environment: Ubuntu 18.04
- Compiler Option: g++
- Total Program Size: 19.7 KB
- Encoding/Decoding mode specified by run-time arguments