Topics in Algorithms Report

== Data Compression ==

2018-26267 김민지

2017-24950 김이은

1. Introduction

This project compares compression rates for various data compression algorithms and discusses their performance. Three given data sets with different characteristics were used in the experiment, and four synthetic data with different sizes and alphabetical numbers were used. In this article, we describe the algorithms and datasets used in this experiment and then show compression ratios in the experimental result section and discussions of them.

2. Review of Algorithms

2.1 Static Huffman coding

Static Huffman coding is the optimal prefix code used for lossless data compression. This method outputs variable-length codes, representing more common symbols with fewer bits.

Huffman coding first calculates the frequency of each character, and creates a priority queue based on that frequency. Then, the algorithm extracts the two elements from the priority queue, create a tree, and build the priority queue again. When this process is repeated and the huffman tree is completed, prefix codes are generated by assigning 0 and 1 to the edges, respectively.

2.2 Adaptive Huffman coding (FGK)

This technique based on Huffman coding was designed by Faller, Gallager and Knuth. The difference with Huffman coding is FGK has no initial knowledge of source distribution, so it allows only one-pass over the data and dynamically adjusts the Huffman's tree as data are being transmitted.

In a FGK Huffman tree, 0-node is used to identify a new character. For past characters, just output the path of the data in the current Huffman's tree. At this stage, if necessary, we need to adjust the FGK Huffman tree and finally update the frequency of relevant nodes.

2.3 Golomb codes

Golomb coding is a lossless data compression algorithm, invented by Solomon V. Golomb in 1960.

Golomb coding divides the input value by parameter m into two parts, a (q) and a remainder (r). The part corresponding to the quotient is represented by unary, and the part corresponding to the remainder is represented by binary prefix code. The code used in this assignment uses Rice coding, a subset of Golomb coding, which represents the remainder as a binary code of length

m, not as an optimized prefix code. Therefore, there may be a lot of Golomb coding that performs better than the algorithm used in the experiment.

2.4 Tunstall codes

Tunstall coding is a lossless data compression technique and a variable-to-fixed length code.

The algorithm requires a distribution of probabilities for each character input and also chooses a target output length n. First, it forms a tree with a root and m children with edge labeled with initial symbols. While the number of leaves is $< 2^n - m$, it finds the leaf with highest probability and expand it to have m children.

2.5 Arithmetic coding

Arithmetic coding is a form of entropy encoding used in a lossless data compression. This algorithm encodes the entire message into a single number, an arbitrary precision quotient q between 0 and 1.

Arithmetic coding calculates the cumulative probability according to the frequency of each character and limits the range of the number to the corresponding precision each time a particular character appears in the input message. If the final encoding between 0 and 1 is decided, it can be decoded according to the frequency of the character.

2.6 LZ77

LZ77 is a lossless data compression algorithm published with LZ78. LZ77 algorithm achieves compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream.

The encoder examines the input string through a sliding window. The window consists of two parts : 1. A search buffer that contains a portion of the recently encoded sequence. 2. A look-ahead buffer that contains the next portion of the sequence to be encoded. This algorithm finds the longest match in the window for the lookahead buffer while the lookahead buffer is not empty. If a match is found, output the pointer P and move the coding position L bytes forward. However, if a match is not found, output a null pointer and the first byte in the lookahead buffer. In this case, move the coding position one byte forward.

2.7 LZ78

LZ78 was published with LZ77 in papers by Abraham Lempel and Jacob Ziv in 1978, and it is also known as LZ2. It constructs the explicit dictionary in order to store the code word.

The LZ78 algorithm reads the character and verifies whether it is in the dictionary in order to encode it. If it already exists in the dictionary, it reads the trailing character and checks whether the string(concatenated characters) is in the dictionary. If the string does not exist in the dictionary, then the algorithm newly add it. In the encoded output, an index with the corresponding string in the dictionary and the next character are written in a file as a pair.

2.8 LZW

LZW was published by Welch in 1984 as an improved implementation of the LZ78 algorithm.

The LZW algorithm initializes the dictionary to contain all strings of length one. While reading a input string, it finds the longest string W in the dictionary which matches the current input. When

such a string is found, the index for the string without the last character is sent to output, and W followed by the next symbol is added to the dictionary. The last input symbol is then used as the next starting point to scan for substrings.

2.9 LZSS

LZSS is the abbreviation of Lempel-Ziv-Storer-Symanski, and is a lossless data compression algorithm, a derivative of LZ77 method. It is one of the dictionary based encoding methods, which attempts to replace a symbol(or a string of symbols) with a reference to a dictionary.

The LZSS algorithm is basically similar to the LZ77, but it is designed to solve the inefficiencies of the LZ77 algorithm. In LZ77, when pointer and offset are small, the length of the corresponding part is only 1 ~ 2 bytes even if decompressed. In this case, it is rather inefficient because the code is expressed in a larger size in the result of the compression. In order to prevent such a phenomenon, the minimum length is set in the LZSS, and compression is attempted only when a pattern exceeding the minimum length is found.

2.10 Common compressors (gzip, bzip2, xz)

gzip is one of the most commonly used file format and a software application used for file compression. This algorithm is basically based on a Deflate algorithm, which uses LZ77 and Huffman coding together. The Deflate algorithm is basically a compression method that uses LZ77 of 32KB window and uses Huffman coding to compress the repeated index (pointer, offset) once again. It is one of the three standard compression formats used for HTTP compression, and usually does not utilize redundancy between files because it is usually used to compress a single file.

Bzip2 is a open-source file compression program, which uses Burrow-Wheeler algorithm as a compression method. Bzip2 mostly shows higher compression ratio than older methods such as LZW and other algorithm using the Deflate method, but slightly slower. bzip2 divides the input message into blocks of 100 to 900 KB, and uses Burrow-Wheeler algorithm to relocate frequently appearing characters together. Then, it transforms the data into a small set of numbers using the move-to-front method and compress using Huffman coding.

xz is a lossless data compression program and a compressed file format at the same time. This program incorporates LZMA and LZMA2 algorithm. LZMA is Lempel-Ziv Markov chain algorithm, and is a dictionary-based compression method. It is somewhat similar to LZ77 algorithm, but maintaining variable dictionary size up to 4GB. LZMA generally shows high compression ratio than bzip2 while decompressing the encoded messages in a short amount of time.

3. Code Description

The code description is based on the pseudo code we wrote, and we did not want to include the raw code capture images. Especially, the codes used for bit I / O and additional functions are often long, and the encode method is difficult to understand. In these cases, we want to explain it by explaining main methods used in the algorithms.
The repository of this project is below: https://github.com/minz95/DataCompression

3.1 Static Huffman coding

```
int main() {
    calculate frequencies
    BuildTable(frequencies)
    WriteBits(table[index], outfile)
}
```

Calculate frequencies
- calculate frequencies of each character appears in the input message.

BuildTable(frequencies)
- build a table (huffman tree) according to frequencies calculated above.

WriteBits(table[index], outfile)
- In the output file, write bits of the character in the input file according to the table constructed above.

3.2 Adaptive Huffman coding (FGK)

```
int main(){
    encode(fp_in, fp_out);
    createTree();
    while(reading a symbol){
        if(new symbol){
            addSymbol(currByte, &zeroNode, symbols);
            updateTree(newNode, root);
        }
        else updateTree(newNode, root);
    }
}
```

encode(fp_in, fp_out)
- This function has two parameters named “fp_in” and “fp_out”.
(“fp_in” : input file, “fp_out” : output_file)

createTree()
- This function makes a default tree with zero node as root and returns a node pointer.

addSymbol(currByte, &zeroNode, symbols)
- if each symbol is a new character while reading a symbol, it needs to be added to the zero node of the tree.

updateTree(newNode, root)
- Whenever adding a new node to the tree, update the tree to satisfy sibling property.

3.3 Golomb codes

```
int main() {
    calculate unary part
    BitFilePutBit(1, outfile)
    Output the bit '0'
    calculate binary part
    BitFilePutBit(binary, outfile)
}
```

Calculate unary part
- unary part is calculated by bitwise shift operation. (unary >>= k)

BitFilePutBit
- This method is used to write a bit ‘0’ and ‘1’ in the output file. The first parameter determines what bit to write, and the second parameter determines where to write the bit. It is also used to write a whole binary encoded codes.

Calculate binary part
- binary part as well is calculated through bitwise shift and mask & operation. (& operation with the bit mask)

3.4 Tunstall codes

```
int main(){
    TunstallTree(); //make Tunstall Tree
    TraverseTree(); //make encoding mapping relation
    encode(); //encode input file
}
```

TunstallTree(root, domain, n_input)
- This function makes a Tunstall tree starting at a root node. We can set a n_input variable which means the target output length (n).

TraverseTree(root, codes)
- Through traversing a Tunstall tree, it creates a mapping relationship between words and encoding output.

encode()
- This function makes a encoding output file from input file based on a mapping relationship between words and encoding output.

3.5 Arithmetic coding

```
int main() {
    calculate frequencies
    calculate range
    calculate high and low value
    Output bits
}
```

Calculate frequencies
- calculate frequencies of each character appears in the input message.

Calculate range
- calculate high-low in order to figure out the range value

Calculate high and low value
- recalculate high and low value according to character in the input file.

Output bits
- output the final encoded bits to a file.

3.6 LZ77

```
int main() {
    compress(in, out, m, l);
    match(window, tree, root, p);
}
```

compress(in, out, m, l)
- in : input file, out : outfile, m : window size, l : look-ahead buffer size

match(window, tree, root, p)
- in compress function, match function finds the longest match in the window for the lookahead buffer.

3.7 LZ78

```
ht_dictionary{
    root
    curr_node
    Next_node
    threshold
    size
};

int main() {
    read a character
    compress_byte(character)
}
```

ht_dictionary
- data structure used for storing the characters(string) and a corresponding code word. Several methods are provided to support operations on a dictionary. (ex. Insert, deletion, search)

compress_byte
- search for a string in the dictionary. If it does not exist, then add a character to the string and continues to search the dictionary. If it exists, output the code word bits and continue to read input file.

3.8 LZW

```
int main() {
    compress(inputFile, outputFile);
    dictionaryInit();
    while ((character = getc(inputFile)) != (unsigned)EOF) {
        if (dictionary contains prefix+character){
            prefix = index;
        }
        else{
            dictionaryAdd(prefix, character, nextCode++);
        }
    }
}
```

dictionaryInit()
- The LZW algorithm first creates a dictionary based on the input file.

dictionaryAdd(pefix, character, nextCode++)
- While reading the character one by one, read the next character if the prefix + character is included in the dictionary, otherwise add it to the dictionary.

3.9 LZSS

```
int main() {
    read character from file
    while(r < bufferend) {
        Output codeword
    }
    flush_bit_buffer()
}
```

output codeword
- If the read characters are in a limited size, then find the code in a dictionary and output the code words.

flush_bit_buffer()
- In the end of the algorithm, flush the bit buffer, in order to write the rest of the bits in the output file.

4. Experiments

This experiment was carried out with minor modifications (input, output, error fixation etc) based on the codes given in the Citation section below. Depending on the degree of optimization of the code, performance may be slightly affected, which is not reflected in the discussion. We can implement the experiment more equitably by implementing it in the same way.

4.1 Environment

OS: Linux
CPU: E5-2630 2.40GHZ
gcc version: 5.4.0

4.2 How to run the codes

Compile command:

make all

Run command for each algorithm:

make ARGS=”-c <infile-name> <outfile-name>” huffman
make ARGS=”<infile-name> <outfile-name> -c” fgk
make ARGS=”-c -k <length of binary portion> -i <infile-name> -o <outfile-name>” golomb
make ARGS=”<infile-name> <outfile-name>” tunstall
make ARGS=”-c -i <infile-name> -o <outfile-name>” lz77
make ARGS=”-i <infile-name> -o <outfile-name>” lz78
make ARGS=”c <infile-name> <outfile-name>” lzw
make ARGS=”e <infile-name> <outfile-name>” lzss

4.3 Dataset

The following is a description of what characteristic each dataset used in the experiment has. In this experiment, data was not preprocessed separately, but was read and processed in byte units.

4.3.1 dnaki

dnaki is simple data in which the letters A, G, T, and C are repeated. Since the number of alphabets is small (only four), redundancy is large.

4.3.2 xmlki

xmlki is a xml data with high redundancy. As we can see in the file, ‘<author>’, ‘<title>’, ‘<year>’, and other words are repeated continuously.

4.3.3 englishki

englishki is an article written in English. Compared to the dataset in 4.3.1 and 4.3.2, this data shows less redundancy. However, due to the nature of English writing, there is still room for compression because there are repeated suffixes and common words.

4.3.4 sd1

sd1 data has relatively small size (32KB), but has at most 255 characters. As a result, the redundancy rate of this data is lower than any other data used in the experiment.

4.3.5 sd2, sd3

These two datasets have moderate redundancy compared to the rest (sd1, sd4). sd2 has higher redundancy than sd1, but still lower than sd3. sd3 has higher redundancy than sd2, but lower than sd4.

4.3.7 sd4

sd4 contains at most 4 alphabets, and the size of the file is 2MB. Therefore, it is considered to have high redundancy.

5. Result

	Static Huffman	FGK	LZW	LZSS	Arithmetic
dnaki	1.84	1.83	1.89	1.19	2.04
enlgishki	1.72	0.55	1.93	1.41	1.73
xmlki	1.25	0.53	1.86	2.19	1.25
sd1	1.00	1.03	0.69	0.89	1.00
sd2	1.00	1.00	0.80	0.67	1.00
sd3	0.99	0.98	0.86	0.49	1.00
sd4	0.89	0.89	0.91	0.55	0.99

<Tunstall code (n = target output length)

	5	10	15
dnaki	0.73	0.73	0.73
enlgishki	1.24	1.41	1.41
xmlki	0.89	0.96	0.96
sd1	0.13	0.13	0.13
sd2	0.75	0.83	0.83
sd3	0.8	0.8	0.8
sd4	0.33	0.33	0.33

<Golomb code (m = length of binary portion)>

	2	3	4	5	6
dnaki	0.20	0.32	0.43	0.5	0.5
enlgishki	0.32	0.53	0.77	0.93	1.02
xmlki	0.27	0.45	0.66	0.84	0.94
sd1	0.23	0.41	0.64	0.1	0.12
sd2	0.56	0.79	0.92	0.92	0.86
sd3	0.84	0.87	0.79	0.67	0.57
sd4	0.62	0.5	0.4	0.33	0.29

<LZ77 (m =Window size, l = Lookahead Buffer size)>

	(4096, 256)	(8192, 256)
dnaki	1.15	1.20
enlgishki	1.42	1.56
xmlki	2.45	2.72
sd1	0.48	1.99
sd2	0.49	1.1
sd3	0.49	0.52
sd4	0.54	0.56

<Common Compressors>

	bzip2	gzip	xz
dnaki	1.93	1.77	2.04
enlgishki	3.37	2.58	3.24
xmlki	6.53	4.65	5.87
sd1	0.99	1	1
sd2	0.99	0.99	0.98
sd3	0.98	0.88	0.97
sd4	0.92	0.86	0.92

6. Discussion

6.1 Optimal algorithm for each dataset

6.1.1 dnaki

- Arithmetic, xz

The characteristic of dnaki is that the number of alphabets is limited to four with very high redundancy, and at the same time, the frequency of each alphabet is biased. When we count the cumulated occurence of each alphabet, we can see that the number of occurence is skewed to A and T. (T: 166562, G: 95237, C: 92344, A: 167141) Due to this skewed frequency feature, we think that arithmetic coding shows the highest compression ratio. In the case of xz, it is a program known to have a higher compression ratio than other programs, and showed good efficiency in compressing dnaki data.

6.1.2 xmlki

- bzip2

xmlki is the most efficient data in commercial compression programs. Unlike other data, there is a big difference in performance compared to single compression algorithm codes with commercial programs because there are many '<>' expressions in xml data, and patterns (ex. <title>, <author>) are repeated many times. It can be seen that the commercial programs show a strong performance in this data.
Moreover, bzip2 shows higher compression ratio than xz algorithm, in the case of compressing xmlki. Since there are repeated patterns in the data, it seems that Burrow-Wheeler transform makes the compression much more efficient.

6.1.3 englishki

- bzip2

Like xmlki data, englishki data is composed in English vocabulary. It seems that the Burrow-Wheeler transform plays a major role in compressing the English words because there is a tendency for specific alphabets to appear near each other. Also, due to the nature of English language, there are many repeated common words and phrases in the input file, and this tendency gives more rooms for high compression ratio. The compression ratio is relatively lower than the case of xmlki, because normal articles have less redundancy than xml files.

6.1.4 sd1

- fgk

The fgk algorithm has a space complexity of $|\Sigma|$, which means the number of types of alphabets used. In the case of sd1 data, at most 255 alphabets are used, but because of the small file size, fgk is expected to show sufficient compression ratio. In fact, the fgk algorithm shows good performance in dnaki data, where the number of alphabets is limited to A, T, G, and C. Also, since the fgk algorithm has less redundancy constraints than other algorithms, it is considered that sd1 with the smallest redundancy can be best compressed.

6.1.5 sd2, sd3

- Static huffman, Arithmetic

sd2 and sd3 share a similar feature, although the number of alphabets and file size are different. Although there is a slight difference in character redundancy, it can be explained that the compression ratio shows a similar pattern. Both data showed no significant compression ratio differences in the static huffman, fgk, and arithmetic algorithms, but Arithmetic algorithm showed the best performance in a negligible difference. (the second decimal place) In many cases, compression results were even larger, and commercial algorithms also did not compress properly.

6.1.6 sd4

- Arithmetic

For sd4 data, we could not get a good compression ratio through most algorithms. Since there are 4 types of alphabet in 2MB, we expected redundancy to be high. However, it seems that pattern redundancy is not well formed because each character is evenly distributed when data is generated.

6.2 Optimal parameters

6.2.1 Tunstall code

In the tunstall algorithm, we can adjust the parameter value called n (target output length). In

this project, we adjusted the n values to 5, 10, and 15. 15 and above did not proceed because the encoding took a very long time. However, there was no significant difference in compression ratio at n = 10 and 15. Overall performance was very good when n = 5. For the sd1 data, which is the smallest size, we set the parameter to n = 20, and after the experiment, we did not find a significant difference from n = 10 and 15.

6.2.2 Golomb code

Golomb coding was not compressed properly in almost all cases. In the case of the englishki data, the best compression performance was achieved at m = 6 and the compression ratio exceeded 1. However, in the remaining cases, the compression ratio did not exceed 1. dnaki, englishki, and xmlki data have the highest compression rate when the parameter is 5 to 6 (binary portion length), while the synthetic data have more compression efficiency as the parameter changes from 4 to 2. (sd1: m=4, sd2: m=4, sd3: m=3, sd4: m=2)

Considering that the file size increases from sd1 to sd4 and the number of alphabets decreases, it is considered that the smaller the value of m is, the more advantageous it can be encoded by using the smaller bits. Golomb coding shows the best efficiency when the alphabet followed the geometric distribution, but not in the given data (dnaki, xmlki, although the frequency was slightly skewed, but not geometric, sd1~4 shows uniform frequency distribution) We can explain that englishki showed the best performance among the given data because it had some of such tendency.

6.2.3 LZ77

There are two parameters in LZ77: window size(m) and look-ahead buffer size(l). In the original code, the default setting is m = 4096 and l = 256. The experiment was carried out by changing the values of m and l several times in the project. As a result, it was observed that the result of m = 8192 and l = 256 is generally the best. Especially, sd1 and sd2 data showed remarkable performance improvement.

6.3 Reasoning for the results

6.3.1 dataset with high compression ratio

- dnaki, xmlki

Both dnaki and xmlki have large redundancy. If pattern redundancy is important, xmlki is compressed to better performance, and if character wise redundancy is important, then dnaki seems to be compressed to better performance. When frequency distribution is important, such as Tunstall coding and Golomb coding, exceptionally, englidhki data is compressed to the best performance, but in most cases the former is established.

One thing to note is that if the pattern redundancy is important and xmlki is compressed with the best performance, the compression performance of englishki is better than dnaki.

6.3.2 dataset with low compression ratio

- sd1

sd1 is the data with the worst compression performance. Of course, the common compressors compressed sd1 with higher compression ratio than sd4, but for the rest of the algorithm, they were not compressed properly, but rather became larger. Because of the large number of alphabetic characters compared to the file size, compression algorithms that take redundancy into account obviously result in poor compression performance. In the meantime, as we increased

the window size to 8192 in LZ77, sd1 showed a compression ratio of 1.99, which is a remarkable result.

7. Contribution

Name	Contribution
Minji Kim	Static Huffman Coding, Golomb Coding, Arithmetic Coding, LZ78, LZSS, bzip2
Yieun Kim	Adaptive Huffman Coding (FGK), Tunstall Coding, LZ77, LZW, gzip, xz

References

[1] Vitter's original paper: J. S. Vitter, "Design and Analysis of Dynamic Huffman Codes", Journal of the ACM, 34(4), October 1987, pp 825–845
[2] Huffman, D. (1952). "A Method for the Construction of Minimum-Redundancy Codes" (PDF). *Proceedings of the IRE*
[3] Ziv, J.; Lempel, A. (1978). "Compression of individual sequences via variable-rate coding" (PDF). *IEEE Transactions on Information Theory*

Code Citations

[1] Static Huffman : http://left404.com/wp-content/uploads/2011/10/simplehuffman.c
[2] FGK : https://github.com/ArthurEmidio/fgk
[3] LZ77 : https://github.com/neoben/LZ77
[4] LZ78 : https://github.com/evilaliv3/lz78
[5] LZW : https://github.com/radekstepan/LZW
[6] LZSS : https://gist.github.com/davidreynolds/3025423
[7] Tunstall :https://github.com/hackerghost93/Algorithms/blob/master/Tunstall%20encoding.cpp
[8] Golomb : https://github.com/MichaelDipperstein/rice
[9] Arithmetic : https://github.com/burtgulash/fav2011-pc-aric