Huffman Encoding

Information Theory - 2018/2019

Manuel Henriques 42546, Sebastião Pamplona 42735

1. Introduction

"One important method of transmitting messages is to transmit in their place sequences of symbols." Using David A. Huffman's algorithm to construct minimum-redundancy codes, we created a program with Java that compresses and decompresses any type of digital file.

2. Overview

We divided our program in two major classes: Compressor and Decompressor. There are also two different approaches to process a file.

1. We dynamically create the probabilities for each symbol of the file and create a header with that information, ensuring that the Decompressor can decompress the file without former knowledge on each symbol or probabilities.
2. With the second approach, both the Compressor and the Decompressor have access to known a priori probabilities and possible symbols, and, as such, there is no need to create a header.

However, there are procedures that both approaches must execute. First, the Compressor creates a Huffman coding tree with the known probabilities in order to replace each symbol with each code. Finally, independently of the way in which the Decompressor acquired the tree, it is also capable of reading the tree to decode the file.

3. Specification and Arguments

3.1 Compressor

The compression is made in the following steps: (Note: step 1 and 5 only apply when the symbol probability distribution is not known a priori).

1. The compressor parses the original file, *Symbol Length* by *Symbol Length*, in order to create a symbol probability distribution. Otherwise, the compressor has access to a file, specifying the probability distribution[1].
2. The Huffman tree is created, accessing the symbol probability distribution.
3. Upon creating the Huffman tree, we map each symbol to its respective code.
4. After that, the actual compression take place. The original file is parsed, changing each symbol by its code.
5. Afterwards, the header is created (see point 3.1.3 for header specification), and added to the head of the compressed file, obtained in step 4.
6. Finally, the result is written into a new file, for later decompression.

3.1.1 Huffman Coding Tree Creation

As we know, the symbols that were coded can be found at Leaf nodes, meaning that the traversal from the root of the tree to a Leaf node (i.e., sequence of *0's* and *1's*) is the symbol's respective code.

The Huffman tree is created in the following steps:

1. The compressor starts by creating a Leaf node for each symbol, with its respective probability associated.
2. The Leaf nodes are stored into a PriorityQueue, which orders the nodes according to their probability (i.e., the node with lowest probability is at the head of the queue).
3. While the queue is not empty, the two least probable nodes are merged into a

[1] Note that this file only applies for symbols with *Symbol Length* equal to 8 bits (i.e., alphabet characters), and when the original file does not contain punctuation.

new node, with the sum of their probabilities associated, and stored into the queue.

4. Once the queue is empty, the Huffman tree is created, with its root being the node resulting from the last merge.

3.1.3 Header

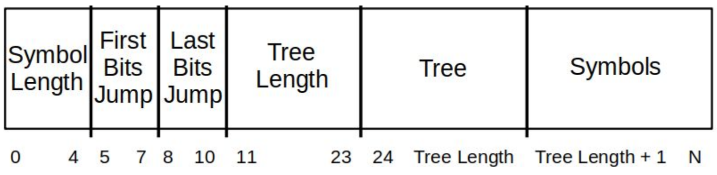

Fig.1 - Header

- *Symbol Length* - represents the symbol length, in bits, used to compress the file.
- *First Bits Jump* - bit gap (number of bits) between the header and the compressed file, in case the bit size of the header is not multiple of 8.
- *Last Bits Jump* - number used to skip from 0 to 7 bits at the end. If the size of the compressed file (without the header) is not a multiple of 8, this stops us from decoding "garbage" at the end.
- *Tree Length* - bit length of the depth first search of the Huffman tree. We used only 13 bits to represent its size because, the larger *Symbol Length* used in our tests was 12 - in the rare case that all symbols were different, the size of the tree would be $2 \times 2^{12} - 1$.
- *Tree* - depth first search of the Huffman tree.
- *Symbols* - the symbols, in the order of the depth first search.

3.2 Decompressor

Before the decompressor can actually start decompressing the compressed file, it needs to create the respective Huffman tree, in order to map each code to its respective symbol. In case the symbol probability distribution is known a priori, the decompressor creates the Huffman tree the same way the compressor does (as was explained in point 3.1.2).

Otherwise, the Huffman tree is created from a header, read from the compressed file. (Note: As was said in step 3.1.3, the depth first search of the original Huffman tree was present in the header). The Huffman tree creation is done by analysing the depth first search (dfs) sequence of *0's* and *1's*. When traversing the dfs, the decompressor needs to decide when to create a node, and if that node is a Leaf. The decision is made with the aid of a Stack, and by comparing the current number of the dfs and the previous, according to the following table:

Previous	Current	Decision
0	0	Push a Node into the stack
0	1	Add a left Leaf to the Node in the head of the stack
1	0	Push a Node into the stack
1	1	Add a right Leaf to the Node in the head of the stack, and then pop that Node

Table 1. - Algorithm for recreating the Huffman tree, from the dfs present in the header.

Because the Leaf nodes are added according to the order they are found through the dfs, their respective symbols can be retrieved by reading the right chunk of bits of the *Symbols* part of the header, because the decompressor keeps track of how many Leaf nodes were added to the tree.

3.2.2 File Decoding

After the recreation of the Huffman tree, the decompressor is able to decode the compressed file.

As the decompressor iterates through the compressed file, it traverses the Huffman tree, following the directions of the bits (i.e., going left or right, according to a *0* or a *1*). When a Leaf node is found, its respective symbol is appended to the decompressed result, and the decompressor goes back to the root of the tree.

3.3 BitSequence

In order to abstract ourselves from bit to bit operations, we created our own class. This class makes it possible for us to code without worrying about adding or retrieving bits.

We can create symbols with any given size of bits, we can concatenate symbols with each other and even create a bit tree to be stored in the header. The main functionalities of this class are:

- **Add bit**: adds a bit to a sequence of other bits. It can be set to 1 or 0 and increments the size of the sequence.
- **Get bit value:** retrieves the value of the bit in a given position.
- **Concatenate:** concatenates two instances of this object, creating a new sequence of bits.
- **Warp:** if we already have a byte array (eg. reading a file from buffer), we can save a reference of that array in a new instance of our class. After warping an array, we can retrieve bits from a selected position of that sequence.

4. Evaluation

We tested our project by compressing different file extensions, focusing on text files to analyse its performance (the other extensions were just to prove that we were able to compress more than just text files). To evaluate the compressor, we measured the compressing time and rate, as well as the decompressing time. We compared these metrics in the two possible cases: when the symbols probabilities are known a priori *vs* when they are not.

4.1 Experiments

4.1.1 Compression with Header

Each file was compressed with different symbol lengths, varying between 6 and 12 bits. We also tested with 5 different text files named:

- “1.txt” with 1 000 bytes.
- “2.txt” with 5 000 bytes.
- “3.txt” with 50 000 bytes.
- “4.txt” with 100 078 bytes.
- “5.txt” with 500 394 bytes.

And with 2 image files named:

- “1.gif” with 355 854 bytes.
- “6.png” with 5 903 837 bytes.

4.1.2 Compression without Header

To test the compression and decompression without header, we used a probability distribution of the english language. Unfortunately, this probability distribution only contained lower case letters and a space symbol. To avoid bias in our results creating random probabilities to add more characters, we removed every special character and upper case letters from the 5 text files. We called this text files:

- “11.txt” with 974 bytes.
- “12.txt” with 4 857 bytes.
- “13.txt” with 48 583 bytes.
- “14.txt” with 97 236 bytes.
- “15.txt” with 486 188 bytes.

Because we used UTF-8 characters from the english language, every file was compressed and decompressed using 8 bits.

4.2 Results

4.2.1 Compression with Header

The results for this topic can be found in topic 6.1. With our results, it is clear that the best symbol length to compress text files is 8 bits. It was to be expected because each character is normally represented in 8 bits.

The compression size for the image files was not considered because it was expected for the file to get bigger with our compression. Image files are usually already compressed, so we only used this files to prove that: 1) our decompressor obtains a file equal to the original (it is easier to compare image files than text files) and 2) to measure the performance of our code.

4.2.2 Compression without Header

The results without header were pretty similar to the results using header and a symbol length of 8 bits. The time and compression were better, but not by a large margin.

4.3 Improvements

Huffman	Compressor	Decompressor
Generate Probabilities	32.91%	-
Create Huffman Tree	0.02%	0.02%
Parse File	66.44%	98.45%
Generate Header	0.02%	-
Write	0.61%	1.37%

Table 2. - Profiling of our Code running file with 5MB and 8 bit symbol length

We started profiling in order to discover a bug, but it ended up giving us more information on how to improve the performance of our implementation. We were able to reduce the time of compressing a 500KB file from 8 minutes to 720 milliseconds.

As we can see, the two major tasks are generating the symbol distribution probability and parsing the files. As a possible future improvement, we considered implementing our project with a C library called Cilk, which is used for multithreaded parallel computing. For generating the probabilities, we could split the file in *t* chunks, according to the number of cores of the processor being used, and process all of them in parallel, merging the results after each one is done. Similarly, the same strategy could be applied for compressing and decompressing the file, if the right precautions took place.

5. Conclusion

After analysing the results, we concluded that the overhead of the header was not proportional to size of the file, both in terms of time and space. With this being said, we can state the its overhead is worth it because the compression becomes detached from the type of file that is being compressed, and more generalized in the sense the there is no need for a file containing the symbol probability distribution.

6. Attachments

6.1 Compression and Decompression with header

In this section, we have the performance of our compressor and decompressor, in the case where the symbol probability distribution is not known a priori, hence needing a header.

Symbol Length	6	7	8	9	10	11	12
Compression Time (ms)	82	5	4	5	3	6	3
Decompression Time (ms)	20	5	1	2	1	1	2
Compressed Size (bytes)	958	1 067	594	1 200	1 091	1 350	943
Compressed Rate (%)	95.80	106.70	59,40	120,00	109,10	135,00	94,30
Header (bytes)	58	131	52	355	356	598	353
Header (%)	6.05	12.28	8.75	29.58	32.63	44.3	37.43

Table 4. - Text file with 1KB

Symbol Length	6	7	8	9	10	11	12
Compression Time (ms)	36	10	9	14	12	15	11
Decompression Time (ms)	15	5	3	4	4	5	3
Compressed Size (bytes)	4 576	4 818	2 784	4 814	4 316	5 148	3 632
Compressed Rate (%)	91,52	96,36	55,68	96,28	86,32	102,96	72,64
Header (bytes)	62	137	62	478	521	1 160	577
Header (%)	1.35	2.84	2.23	9.93	12.07	22.53	15.89

Table 5. - Text file with 5KB

Symbol Length	6	7	8	9	10	11	12
Compression Time (ms)	179	118	54	92	60	70	72
Decompression Time (ms)	46	23	20	28	23	28	25
Compressed Size (bytes)	45 186	47 102	27 307	44 256	38 770	41 111	31 550
Compressed Rate (%)	90,37	94,20	54,61	88,51	77,54	82,22	63,10
Header (bytes)	63	143	62	555	638	1 183	743
Header (%)	0.14	0.3	0.23	1.25	1.65	2.88	2.35

Table 6. - Text file with 50KB

Symbol Length	6	7	8	9	10	11	12
Compression Time (ms)	241	124	110	134	139	138	146
Decompression Time (ms)	67	45	40	49	48	55	47
Compressed Size (bytes)	90 364	94 110	54 576	87 901	76 965	82 944	62 355
Compressed Rate (%)	90,29	94,04	54,53	87,83	76,91	82,88	62,31
Header (bytes)	64	143	63	555	658	1 705	750
Header (%)	0.07	0.15	0.12	0.63	0.85	2.06	1.2

Table 7. - Text file with 100KB

Symbol Length	6	7	8	9	10	11	12
Compression Time (ms)	983	699	720	567	571	699	525
Decompression Time (ms)	245	252	231	227	240	233	201
Compressed Size (bytes)	451 604	469 982	272 632	437 885	382 285	408 130	308 971
Compressed Rate (%)	90,25	93,92	54,48	87,51	76,40	81,56	61,75
Header (bytes)	64	143	63	567	663	1 732	761
Header (%)	0.01	0.03	0.02	0.13	0.17	0.42	0.25

Table 8. - Text file with 500KB

Symbol Length	6	7	8	9	10	11	12
Compression Time (ms)	947	828	642	618	714	674	758
Decompression Time (ms)	308	221	215	197	215	215	232

Table 9. - Gif file with 356KB

Symbol Length	6	7	8	9	10	11	12
Compression Time (ms)	17275	17376	12204	12417	12212	10152	9837
Decompression Time (ms)	5238	4556	5102	4540	4028	5355	4654

Table 10. - PNG file with 5 903KB

6.2 Compression and Decompression without header

In this section, we have the performance of our compressor and decompressor, in the case where the symbol probability distribution is known a priori, hence not needing a header.

Files	1.txt	2.txt	3.txt	4.txt	5.txt
Compression Time (ms)	37	18	75	100	408
Decompression Time (ms)	24	14	49	82	257
Compressed Size (bytes)	974	2 522	25 257	50 520	252 602
Compressed Rate (%)	52,46	51,93	51,99	51,96	51,96

Table 11. - All the 5 text files