

## Data – Compression

Despite growing capacity and falling prices of storage devices, data compression is still needed for two main reasons. One is that more data is collected than ever before, both in the scientific and in the economic sector as well. Companies accumulate huge amounts of data about their customers and they often have to back up their data on a daily basis or even more frequently. Especially if the collected data doesn't need to be accessed frequently it is far more efficient and cost effective to compress the data than to buy larger and larger storage devices. The second and probably most important application for data compression is in networking. The capacity of networks, especially the internet, is still too low for the enormous sums of data that need to be transported everyday.

The field of data compression can roughly be divided into compression afflicted with losses and loss-free compression. In my proseminar I have dealt with the following four categories of loss-free compression algorithms: Run-length coding, Huffman coding, Arithmetic coding and Lempel-Ziv algorithms.

### Run-length coding

Probably the most naive approach to data compression is run-length coding. The idea of run-length coding is to represent the repeated occurrence of a character in a given string by the character plus the number of occurrences. If we take for instance take the string:

AAABBBCCAAAADDDBBBBBBBBBBBBBBBBEEEEEEEEEEEEEEEEEEEEEDFFFF

We can easily see that instead of writing 'AAA' it would be shorter to write '3A'. In this way the string above can be written as: 3A 3B 2C 4A 3D 12B 18E 1D 4F . Of course it is a matter of implementation if one would represent the single letter 'D' as '1D', which obviously elongates the string instead of shortening it, or simply leave the 'D'. A much more crucial matter is how one would represent the length of the run. Especially if the symbols for numbers also occur as characters in the string. One solution of how to distinguish between symbols representing numbers and symbols representing characters of the string is the usage of a so-called 'escape-symbol'. Then the occurrence of this escape-symbol would signal that the following symbol is a counter, while a symbol without a preceding escape-symbol would represent a character. If we chose 'X' as the escape-symbol for our example and use the  $i$ -th letter of the alphabet as the number  $i$ , the example-string can be encoded as: XCA XCB CC XDA XCD XLB XRE D XDF .

Of course run-length coding of natural-language texts will not yield very good results since the repetition of a character is not very likely and rarely exceeds two characters at a time. So the main application for run-length coding is with binary files and bitmaps. In binary files only two symbols occur viz. 0 and 1. Thus runs of zeros and ones can be expected, which are long enough to ensure a good compression ratio. In binary files we can also exploit the fact that runs of zeros and ones always alternate. So we don't need to explicitly name the character because we always know that a run of zeros will be followed by a run of ones. In bitmap files we often have rather long runs of color-values which can easily be compressed with run-length coding. So a row of 50 black pixels can be coded as 50 00, where 00 is the color-code for black.

## Huffman –coding

Usually text files are coded in ASCII (8 bits per character) or unicode (16 bits per character). Since all characters are encoded with the same fixed number of bits, a code like the ASCII code is called a 'fixed-length code'. Now the question is, why should characters that occur rather frequently use as much space as characters that occur less often. The answer to this is to use a 'variable-length code', i.e. a code where the number of bits used to encode one character varies. The Huffman-algorithm gives us a way to find an optimal code for a given probability of the occurrence of each character in a string.

To construct a Huffman code for a certain string, one needs to know the frequency with which a character in the string occurs. There are two different strategies on how to obtain these frequencies. One is to calculate the frequencies each time a string is processed. This means that generating the code will take longer and the code has to be shipped along with the encoded data for the decoding process. The advantage of this method is that the resulting code will be optimal for the given string. The other method is to use given statistics. One wants to compress an English text for example, then a statistic about the average frequencies of letters in the English alphabet can be used to create the Huffman code. With this method the code doesn't need to be submitted together with the data because both the encoder and the decoder use the same frequencies to generate the code. The disadvantage of this method is of course that the text might differ from the average by far. Thus encoding could in some cases lengthen the text instead of compressing it.

Once the probability of each character is known, whether it has been calculated or results from a statistic, a Huffman code can be generated in the following way: First insert all distinct characters of the string that is to be compressed into a priority queue with the frequency of the character as a key. Then pick the two items in the priority queue with the smallest key, i.e. with the lowest frequency. Combine those two items into a binary tree, where the two characters are the sons and store the sum of the frequencies of both children in the root. Now insert this new binary tree into the priority queue with the value of the root as its key. Then repeat the steps above until there is only one binary tree left in the queue. The resulting Huffman-tree has all distinct characters of the string stored in its leaves and characters with a low frequency are farther away from the root than characters with a high frequency. Now we can assign a zero to each left branch in the tree and a one to each right branch in the tree, or vice versa, yet we need to stick with one way throughout the whole tree. Then we can get the code for a character by traversing the tree from the root to the leaf that stores the character and collecting all zeros and ones on the way. Since all characters are stored at the leaves of the tree, we will get what is called a 'prefix code'. That is a code where no code word is a prefix of another code word.

Now let  $d$  be the number of distinct characters in the string and  $n$  be the length of the string. If the priority queue is implemented in a heap structure, operations (insertion, deletion, etc.) will take  $O(\log d)$  time. For the initialization of the priority queue we need to operate on the queue once for each character in the string, so the initialization will take  $O(n \log d)$  time. In each step of the loop in which the tree is created, the queue is shortened by one element. So the loop will be repeated  $d-1$  times. Thus the running time of the loop is  $O(d \log d)$ . This results in a total running time of  $O((n+d) \log d)$ . Especially if the frequencies for the characters result from a fixed statistic, we can say that  $d$  is just the number of symbols in our alphabet and thus a constant number. This way the running time can be reduced to  $O(n)$ . On the other hand if we want to know how the algorithm depends on the size of the alphabet with a string of a fixed length then  $n$  is constant. Thus the running time of the algorithm is  $O(d \log d)$ .

The reason why the Huffman algorithm yields an optimal code is the following.

Assume we are given a tree  $T$  which represents an optimal code. Let  $w_1$  and  $w_2$  be the two characters of the string represented by  $T$  with the smallest probability, i.e. the two items the Huffman algorithm merges first. Let  $V$  be an internal node of  $T$  with maximum distance from the root. Let  $a$  and  $b$  be the children of  $V$ . If  $a$  and  $b$  are different from  $w_1$  and  $w_2$  then  $w_1$  and  $w_2$  appear at some other position in the optimal tree  $T$ . Now we swap  $w_1$  and  $a$  and swap  $w_2$  and  $b$ . Since  $w_1$  and  $w_2$  are not moved closer to the root by this action the result will still be an optimal tree. By inductive application we can transform  $T$  into a Huffman tree without losing the property of being optimal.

## Arithmetic coding

With the Huffman code each character is assigned a code of  $-\log_2 p$  bits length, where  $p$  is the probability of the character. So a character with probability 0.3 will be assigned a code of 1 or 2 bits since  $-\log_2 0.3 \approx 1.74$ . Yet an optimal solution would be a code with a length of 1.74 bits. Arithmetic coding tries to circumvent this problem by assigning one code word to the entire string instead of one code word per character. So the input string is represented by a number in the interval  $[0,1)$ . The method of how to pick a unique number between 0 and 1 is this: First divide the interval among the distinct characters in the string by assigning each character a subinterval which is in its size proportional to the probability of the character. Then we pick the subinterval which belongs to the character we want to encode and make this our new interval. Now we divide this interval and pick the according subinterval for the next character. If our string consists of three distinct characters  $a, b$  and  $c$  with probabilities  $p_a = 0.1$ ,  $p_b = 0.7$  and  $p_c = 0.2$  then we divide  $[0,1)$  into the three subintervals  $[0,0.1)$ ,  $[0.1,0.8)$  and  $[0.8, 1)$ . If the first character in our string is a 'c' we choose  $[0.8,1)$  as our new interval. (Notice that no matter which characters follow now, the resulting code will be within the interval  $[0.8,1)$ , which is important for decoding.) After the whole string is processed, we can choose any number within the resulting interval as a code for the entire string. Since frequent characters get a larger interval than less frequent ones, a shorter number can be chosen for a string where some characters appear frequently.

Instead of dividing the current interval into subintervals in each step, the new interval for a character can be calculated with this formula:

$$\text{NewTop} = \text{OldBottom} + \text{OldRange} \cdot \text{TopRange}(X)$$

$$\text{NewBottom} = \text{OldBottom} + \text{OldRange} \cdot \text{BottomRange}(X)$$

where  $\text{OldBottom}$  is the lower boundary of the current interval,  $\text{OldRange}$  is the range of the current interval and  $\text{TopRange}(X)$  and  $\text{BottomRange}(X)$  are the upper and the lower boundary of the initial interval of the character  $X$ .

For decoding we again divide the interval  $[0,1)$  into the subintervals for each distinct character in the string. We know right away that the first character of the string will be the character in whose interval the code-number is. Then we reduce the effect of this character on the string with this formula:

$$\text{NewCode} = (\text{OldCode} - \text{BottomRange}(X)) / \text{Range}(X)$$

where  $\text{Range}(X)$  is the size of the interval which belongs to the character  $X$ , i.e. the probability of  $X$ .

## Lempel–Ziv algorithms

In order to ensure optimal codes both with Huffman coding and arithmetic coding the frequency of each character has to be calculated each time a string is processed and the frequencies have to be sent along with the data for decoding. The class of Lempel–Ziv algorithms doesn't use probabilities to generate a code for the characters or for the string but takes a different approach. The idea here is to encode a substring as a reference to the same substring somewhere in the preceding text.

The way this is achieved by the LZ77 algorithm, which was first published in 1977 by Jacob Ziv and Abraham Lempel, is the following. Find the longest prefix  $P$  of the part of the string that is yet to be encoded that occurs as a substring  $S$  within the already encoded part of the string. Then this prefix can be encoded as the triple  $(s, l, c)$  where  $S$  is the starting position of  $S$ ,  $l$  is the length of  $S$  and  $c$  is the character that follows  $P$  in the string i.e. the first character with which the prefix is no longer a substring of the preceding part of the string. If there is a character  $c$  to be encoded which hasn't been encoded before it can be represented by the triple  $(0,0,c)$ . In the LZ77 algorithm and most algorithms derived from the LZ77, the part of the string in which the program searches for a substring is limited to several thousand characters. An example for an algorithm derived from LZ77 would be LZH. Where the references are Huffman–coded. The running time of the LZ77 algorithm very much depends upon the way the methods for finding  $S$  are implemented. A running time of  $O(n)$  can be achieved.

A similar algorithm is LZ78. Though here we no longer have a fixed window in which we look for a substring. During encoding the LZ78 creates a dictionary of phrases which already occurred in the encoded part of the string. It starts out with defining phrase 0 as the null or empty string. Now the first character  $c_1$  of the string can be represented by the pair  $(0, c_1)$  i.e. the empty string plus the first character with which the phrase doesn't appear in the dictionary. Then this new phrase is added to the dictionary. In this manner the whole string is processed. For decoding the dictionary is also initialized with the empty string and grows as more of the string is decoded.

## Bibliography

R. Sedgewick

Algorithms

Addison–Wesley, 1983

(good description of run–length coding and Huffman codes)

M. T. Goodrich, R. Tamassia

Data Structures and Algorithms

John Wiley & Sons, Inc., 1998

(very brief description of the Huffman algorithm)

T. H. Cormen, C. E. Leiserson, R. L. Rivest

Introduction to Algorithms

The MIT Press, Cambridge, Mass., 1990

(thorough explanation of the Huffman algorithm)

D. Salomon

Data Compression

Springer, New York, 1998

(intelligible description of arithmetic coding)

R. N. Williams

Adaptive data compression

Kluwer Academic Press, Boston, 1991

(sporadically confusing text on arithmetic coding but a good overview over LZ–coding)

T. C. Bell, J. G. Cleary, I. H. Witten

Text Compression

Prentice Hall, Englewood Cliffs, 1990

(detailed description of different LZ–algorithms)

D. Gusfield

Algorithms on strings, trees and sequences

Cambridge University Press, 1998

(good introduction to the principle of LZ–coding)

L. P. Deutsch

RFC 1951 DEFLATE Compressed Data Format Specification ver 1.3

<ftp://ftp.uu.net/graphics/png/documents/zlib/zdoc-index.html>

1996

(detailed definition of a lossless compression format that uses both Huffman–code and LZ77 algorithm)