EN

Kayıpsız Metin Sıkıştırma ~ Lossless Text Compression

Bilal Onur Eskili | 24.01.2021

(onur@bilalonureskili.com)

Lossless text compression is possible by replacing the words encountered in the text in certain time with their equivalents that take up less memory space than them. One of the lossless text compression algorithms is Huffman Coding. It tried to give less bit length equivalent to word that has most frequency in text. With this algorithm, it is possible to make bigger compressions. However, for the decompression process, there is a necessity of knowing dictionary that used in compression process. For example, due to new inputs entered in the same database, we must recompress all data again because the first dictionary will be changed with new inputs entered database. There are lots of compression algorithms in the world that satisfies certain needs. The most popular ones are gzip (deflate algorithm), LZ77, LZ78, Shannon-Fano and Brotli.[1]

Preliminary Ideas

First of all, my aim is not only the compression ratio but also time of compression and decompression, the protection of database structure and minimizing the number of elements in dictionary. The main purpose of a compression algorithm is to optimize the compression ratio and compression time. I want to use this algorithm in a database structure, in other words, it is not a specific database like SQL or NoSQL etc. it is a data storage in a certain structure.

As I mentioned before, the main goal of the compression algorithms is to write parts of the content shorter than they are. We can do this by giving specific values to the words that are frequently encountered or likely to be encountered in the text. For example, in compression process of the word “ankara” in Huffman Coding, we can see that the letter a is repeated 3 times we should give it to shortest bit. Here, if we substitute (0) instead of the letter a, (110 and 111) instead of n and k, and (10) instead of r, we will complete our compression with the Huffman algorithm. We can express the word (6 * 8), which we normally express with 48 bits, with only 11 bits (01101110100) with the help of Huffman algorithm.

Type-Based Compression Algorithm

First of all, I want to start with a database includes name and surname columns. Let’s look an example under ideal conditions, in other words, names are correctly spelled and all names (not surnames) are in the dictionary. I will express this database in txt file format. Also,

[1] For detailed information about lossless compression algorithm (http://mattmahoney.net/dc/text.html)

the new line character “\n” indicates new row and the space character “ “ indicates the next column. In Turkish, the names generally consist of two or one name and one surname. If we think one of the lines (row) is X Y Z (X and Y name, Z surname) or X Z (X name, Z surname). So, the database is like X1 Y1 Z1\nX2 Y2 Z2\nX3 Y3 Z3\n. In text file:

X1 Y1 Z1
X2 Y2 Z2
X3 Y3 Z3

The structure is that the last column shows the surname and the remaining columns show name. In 2019, the most popular 6 names in Turkey formed the 12.8 million population out of 83 million. [2] It is not possible to add all names to the dictionary but it is possible to add names owned by most of the population and if we replace these names with some 2 bytes equivalents (2 characters). There is a certain compression process on file. Under ideal conditions, (i.e., dictionary includes all used names and neither of the surnames includes these names in some parts. All name and surname combinations are randomly chosen.

Raw text size	Size after compression	Size after compressed with LZW[3]	Compression Ratio	Compression Ratio with LZW
4.32 KB	2.41 KB	2.40 KB	1.7926	1.8060
8.59 KB	4.81 KB	4.19 KB	1.7859	2.0501
17.15 KB	9.58 KB	7.23 KB	1.7900	2.3753

Compression Ratio = Initial size / Final size

As we see the algorithm that I used more or less gives a compression ratio around 1.79. The reason why it still remains same is all names are in dictionary however the surnames are not and tried to protect the data structure as are. For example, if we have data like:

X Y Z
H F T

And after the compression with my algorithm the result is:

N M Z~K L T

Here, N and M are the shorter equivalents of X and Y, and K and L are the shorter equivalents of H and F than them. The shorter means which is expressed in less bits. The special character, “~” is replaced with new line character. In this example, despite the fact that the size of

[2] https://www.ensonhaber.com/ic-haber/icisleri-bakanligi-2019-yili-dogum-ve-isim-istatistikleri
[3] Lempel–Ziv–Welch

compression rate (for this dataset 1.79) is not increased with the increase of the size of raw data, the algorithm protects the data structure in the database. Protection of data structure is a good way to store data because to exemplify, if we want to get the data in the 2nd row (H F T), we should take the text between first special character (~) and second special character (if there, otherwise to the end of the text) then decompress it. If we compress database with LZW, we must decompress whole data in every request or if we compress lines in database separately with LZW we can do decompression for one line at request but the compression for 1 line (row) is not effective way of compression (look at Appx. 1). In algorithm that I used, for this example is focused on the name and surname, generally name type. For different types, the dataset should be recreated and chosen based on the frequency of elements for the that specific type.

In this case, the disadvantages of the compression algorithm depending on the type I use:

- The compression rate is not high compared to other algorithms for big size data.
- Sensitivity is high (to instance, in a misspelling case the algorithm cannot replace the misspelled word with its equivalent.)
- If we want to two-character (2 bytes) equivalent for the specific word or prefixes in the database that not includes any number, we can have maximum 10*256=2560 equivalent. If database includes numbers, then dictionary can have 256 equivalents for 2 character (1*256).

Advantages:

- The requested row (line) can be taken and decompressed so as to reach the data. This may be possible because the data structure is preserved in the compressed variant.
- The speed of compression process is handled in short time because algorithm directly replace the words/prefixes with their equivalents.

One of the important topics of type-based compression algorithm is choosing the equivalents for words or prefixes. There should be another algorithm for this process. For example, if we try to make a type-based compression algorithm (TBCA) for a newspaper website, we should take all news that has been published before on the website in order to analyze frequencies of the words. But the most crucial point in this process is trending words in certain time or period. For instance, as we know the US president is elected for 4 years (term). However, in the newspaper website probably, the appearance frequency of the former president's name is higher than current president's name. But in the future, the newspaper will publish more news includes current president's name. That is why we should add name of current president before name of the former president to the dictionary.

Case Studies

Another example is identification numbers. In Turkey, everyone has 11-digit identification number (like social security number in US). If we try to compress a column that includes people's Turkish identification numbers, we can compress from 11 bytes to 5 bytes per identification number. For this compression, we should add combinations to the dictionary. In this type of TBCA is more efficient (more compression) than LZW.

Another case is plate numbers. In Turkey, plates are 5-digit 2 letter or 5-digit 3 letter (00ABC000, 00AB000). We can compress from 7 bytes to 4 bytes per plate with 7 characters and from 8 bytes to 5 bytes per plate with 8 characters. In this TBCA, compressed file is smaller than file compressed with LZW (Appx. 2).

Conclusion

Type-based compression algorithm can be used in data types that can be previously calculated approximately frequencies of words like city names, people names, product names, plates, identification numbers etc. Also, this algorithm can be used in online games because there must be fast response from the servers to client. Furthermore, we can use TBCA with LZW algorithms in compatible with each other if there is a necessity of higher compression.

Appendices

1.

Line content	Original (in bits)	Compressed with LZW (in bits)	Compression Ratio (Raw/Compressed)
Ankara Istanbul 600	160	140	1.14
Ankara Bolu 400	128	112	1.14
Ankara Kirsehir 350	160	152	1.05
Istanbul Izmir 400	144	126	1.14

If we directly compressed these 4 lines in once, compression ratio would be: 1.3214

2.

Plate	Original Size	Compressed with TBCA	Compressed with LZW
82ABC888	64 bits	40 bits (added 3 extra bits)	56 bits
93CB753	56 bits	32 bits	49 bits

Keywords

Text compression, compression algorithms, type-based compression algorithm, TBCA, database compression