Optimization Project: Ziv-Lempel Data Compression

Thomas Vicente, Joon Hyun Byon

The nature of the work of modern applied scientists gives rise to the need for minimizing the use of storage resources for the transmission of information. The challenge is to achieve such compact representation of data thanks compression and decompression mechanisms allowing no loss of the original information.

Proposed Algorithm

To address the above problem, we provide an algorithm closely ressembling the Ziv-Lempel version 78. The latter comes from the family of lossless, dictionary-based compression that exploits the structural redundancy of human and machine-generated information. Its purpose is to achieve compression by dynamically mapping a dictionary (set of indices) to sets of recurring patterns. It then stores and transmits the dictionary, instead of the original information.

Mechanism

Our compression algorithm works the following way:

1. Let the object to be compressed (input) be a sequence of alphanumerics and special characters, T, of length n.

2. We initialize an empty dicionary, which will be used to assign, dynamically, a relationship between indexes $I_1, ...I_m$ and a unique pattern p of characters. The index is encoded as (i, c), where i is the index contained in the dictionary associated with with the longest match (prefix) up to the last character of an input and c corresponds to the input's last character. The index value $i = 0$ is used in case there is no match, yet. There is no bound on the amount of indexes or length of sub-strings.

3. Each index corresponds to a 2^k combinations of binary code, where k corresponds to the number of 256 combination tables necessary to code the output. Indeed, for general purposes, we used the ASCII \$2ˆ{8bits}-lengthed index to code any string. The dictionary is built dynamically, as the encoding algorithm scans and compresses the input.

4. The encoding algorithm uses a loop to read the input from the beginning one by one, verifying at each position $T_1....T_n$, whether a pattern p is in the dictionary. As each

position is read in, the encoder starts creating a substring and at each step, checks whether the substring matches with any pattern p at index I_j in the dictionary. In a recurring process, if there is a match, then the offset moves to the next letter, T_{i+1} and checks whether the substring $p \cup T_{i+1}$ exists in the dictionary. If there is no match, then the reading process stops and the pattern $p \cup T_{i+1}$ is assigned a new index, I_k with encoding (p, T_{i+1}). We note that the encoder is greedy, as it is designed to find the longest possible prefix in the dictionary, before deciding whether or not a new substring will be added to the dictionary or not. Furthermore, by referencing new indices to existing ones, decoding is made possible without using a separate decoding dictionary, as long as the decoding process starts from the beginning of the coding sequence.

5. The above process repeats itself until the length of T has been completely visited, by which T would be parsed into a sequence of substrings t, each forming a unique pattern p associated with an index I. Thus, T is encoded as a collection of parsed sub-strings $T = t_1 t_2 ... t_{i-1} \overline{t_i}$. The last substring $\overline{t_i}$, could be a unique or non-unique pattern.

6. Decompression would initiate with the first index I_1 and sequentially reading each index (i, c), where the decoder concatenates pattern p located in position i with character c.

Assessment

The proposed algorithm is able to dynamically construct a dictionary without bounds from locally, observed patterns inherent to the structure of the data. This results in the ability to capture long patterns, and thus, increasing compression. Therefore, we can expect the algorithm to perform well when the entropy relative to the length of the input is low.

On the other hand, because of its boundless property, we have to worry about the number of bits allocated to the index, itself. This would be evident in situations where the input file is highly heterogenous, where the algorithm will not be able to effectively compress the original data.

Proof of Correctness

Let n be the number of characters in the original string.

Let m be the number of unique sequences found by the greedy algorithm.

Let α be the cardinality of the set containing all binary combinations necessary to code all ASCII characters, we usually consider it to be 256.

Let β be the cardinality of the set, belonging to all binary combinations necessary to code all ASCII characters, and new entries.

Let $log_2 \alpha$ be the unit weight in bits of the original text.

Let $log_2 2^k \alpha$ be the unit weight in bits of the uncompressed text, where $k = log_2 \frac{\beta}{\alpha}$.

The algorithm ensures that:

$$\frac{mlog_2 2^k \alpha}{nlog_2 \alpha} \leq 1$$

At worst, we have a compression ratio of 100%. Therefore, there can be equality when the maximum combination reaches a $log_2 2^k \alpha$-bits table.

$$\Leftrightarrow \frac{log_2 2^k \alpha}{log_2 \alpha} \leq \frac{n}{m}$$

$$\Leftrightarrow \frac{k}{log_2 \alpha} \leq \frac{n}{m} - 1$$

This will typically yield a strict inequality when the entropy of our original string is low enough: when m and k are small, relative to n.

Output

For the R code, please refer to the Appendix.

Compression Ratios:

Using the term in the inequality previously described,

$$\frac{mlog_2 2^k \alpha}{nlog_2 \alpha}$$

we compute the following compression ratios:

a) DNA.txt: 25.76%
b) Linear programming text: 42.15%

The below sensitivity analyses shed light on the possible causes for the different compression rates observed between both texts. Although the sequence of the four DNA characters seem to have been generated from a random process (p=0.25, 0.25, 0.29, 0.21), the small number of unique symbols create a higher redundancy than that of the linear programming text, which in turn, leads to higher compression.

Sensitivity Analyses:

These different magnitudes for the observed ratios led us to conduct a sensitivity analyses. It shed light on general causes leading our algorithm to perform more or less well. Although the sequencing of the four DNA characters seems to have been generated by a random process (with probabilities .25, .25, .29, and 0.21), the small number of unique symbols creates a higher redundancy than the one of the linear programming text, which in turn, leads to higher compression for DNA.txt. We believe this confirms the theory predicting that texts with a relative low entropy can be compressed better.

We ran two sensitivity analyses on texts equivalent to DNA.txt to understand the effects that randomness and heterogeneity have on the compression ratios.

a) Effect of randomness on compression ratios We simulated two tests, each with 4 characters, but generated from different distributions. The first was generated with probabilities of 0.1, 0.1, 0.4, and 0.4 respectively, and tested against a perfect random distribution. The following compression reatios were obtained:

- For a non-random sequence: 29.82%
- For a random sequence: 35.92%

Correlated sequences contribute to higher compression.

b) Efect of redundancy on compression ratios We, again, simulated two tests: one originating from a low entropy distribution, against one of high entropy.

- For a sequence of 4 characters: 35.85%
- For a sequence of 8 characters: 57.51%

Redundancy contributes to higher compression.

APPENDIX

```
zl78 <- function(str) {
    #transform one string into multiple elements
    S <- unlist(strsplit(str, ''))
    S <- S[which(S!='\n')]

    #initialize the dictionnary, and the string element to be encoded
    D <- matrix(, 0, 2)
```

```
    T <- ''

    #iterative discovery of sequences, filling a dictionnary
    for (i in 1:length(S)) {
        T <- paste0(T, S[i])
        if (T %in% D[,1]) {
            if (i + length(T) - 1 == length(S)) #add only the last sequence found
                D <- rbind(D, c(T, which(D == substr(T, 1, nchar(T)))))
            else next
        } else { #store all sequences but the last
            p <- ifelse(nchar(T) == 1, 0, which(D == substr(T, 1, nchar(T)-1))) #comput
            l <- substr(T, nchar(T), nchar(T)) #compute code
            D <- rbind(D, c(T, paste0(p,',',l))) #fill the dictionnary
            T <- c()
        }
    }

    #counting the number of bits needed by our code
    bits <- 7 + ceiling(nrow(D)/255)
    rate <- round((nrow(D)*bits) / (length(S)*8), 4)*100

    #based on the following output, the decoder will read the first part of each eleme
    #it will then read the ASCII-256 string after the comma (the first one, when there
    list(code = D[,2], compression_ratio = paste(rate,'%'))
}
```

References

[1] Nelson, Mark, and Jean-Loup Gailly. The Data Compression Book. New York: M & T, 1996. Print.

[2] Sayood, Khalid. Introduction to Data Compression. Amsterdam: Elsevier, 2006. Print.

[3] Shor, Peter. "Lempel Ziv." (n.d.): n. pag. Web. 6 Dec. 2015. http://www-math.mit.edu/~shor/PAM/lempel_ziv_notes.pdf.

[4] "Compression Algorithms: Huffman and Lempel-Ziv-Welch (LZW)" 13 Feb. 2012 http://web.mit.edu/6.02/www/s2012/handouts/3.pdf.