Universidade de Aveiro

Departamento de Eletrónica, Comunicações e Informática

Algorithmic Information Theory (2023/24)

Lab work nº 1

27 Mar 2024

Rafael Pinto, nº 103379

Ricardo Antunes, nº 98275

Pompeu Costa, nº 103294

Contents

Introduction

This report addresses the implementation and exploration of a copy model in context of data compression. Copy model is a strategy for predicting the next symbol in a sequence by referencing previously occurring symbols. In this project, two programs were developed: cpm and mutate. The cpm program implements the copy model and the mutate program mutates the content of a file according to a certain mutation probability. Throughout this report, we will detail the implementation of these programs and conduct an analysis of the results obtained from the tests performed with the *cpm* and *mutate* program.

Implementation

CPM

CPM is a program to “simulate” the compression of files. This program estimates the quantity of bits of the compressed file. CPM uses an algorithm called Copy Model. This algorithm is composed of two sub models. The Repeat Model and the Fallback Model.

Repeat Model

The Repeat Model, given a sequence of characters, tries to guess the next character, based on the past characters of the sequence.

The probability of the model guessing correctly is given by the following equation:

$$P(hit) = (Nh + a) / (Nh + Nm + 2 * a)$$

Where Nh is the number of hits, ie., the number of times the model guessed correctly, Nm the number of misses, i.e.., the number of times the model failed to guess the character, and “a” is a smoothing parameter to avoid division by 0.

If the model is guessed correctly, then the number of bits required to encode the character is given by:

$$Bits = -\log2(P(hit))$$

If, on the other hand, the model failed to guess, then the number of bits is given by:

$$Bits = -\log2((1 – P(hit)) / (alphabet\_size – 1))$$

Where alphabet_size, is the number of unique characters in the given file.

Fallback Model

The Fallback Model sees the last 200 characters of the sequence and based on the frequency of each character it calculates the number of bits required to encode the current character.

To demonstrate this, let us use 5 characters instead of 200 and let us say that those last 5 characters are: A, A, B, C, A. Then the number of bits is calculated by the following formula:

$$Bits = -(3 / 5) \log2(3 / 5) - (1 / 5) \log2(1 / 5) - (1 / 5) \log2(1 / 5)$$

Ie., the number of times each character appears in the sequence divided by the quantity of characters in the sequence, times log2 of that value.

When is each model used

Only one model is used at a time. The Copy Model is used when the ratio between the number of hits and the number of misses is less that a threshold. When that threshold is reached, the Fallback

Model is used instead. The algorithm goes back to the Copy Model if the last k characters of the current sequence have appeared before, which he then uses to try to guess the next character.

Mutate

This program mutates the content of a file based on a given probability. This means, that the program will iterate through each character of the file and according to a given probability, will or will not, change that character to another character in the file.

To reach this objective, we implemented a function *mutateFile*. This function receives, as arguments, the path of the file and the probability to mutate. It creates a file with the mutated content, according to the probability provided. The function starts by opening the provided file in binary mode, allowing the function to read it byte by byte. Then it checks if the file was successfully opened and if it was not then it shows an error message. After that, the output file name is generated by concatenating "_mutate" at the end of the name of the provided file, which is then open to write. To guarantee that the sequence of random numbers generated is different in each execution of the program, we initialize the seed based on the current time. The function then proceeds to go through each character of the file, mutating or not, according to the provided probability. If it chooses to mutate then a random character from the file's alphabet is chosen.

Results

In this section, we will present the results obtained from the tests performed with the *cpm* and *mutate* program and conduct an analysis of them. Our objectives are to observe the effect that input parameters have on the copy model performance, how our algorithm alternate between each model and how the estimated bits are affected when there is more randomness in a file. Additionally, we will also compare how general-purpose compressors, such as gzip, bzip2, and zstd, behave with different levels of randomness in a file.

Different Configurations

We tested our implementation of the copy model algorithm using different parameters combinations for the file *chry.txt*. In this experiment, the window size ranged from 3 to 15, the threshold ranged from 0.3 to 0.9 and the alpha was 1. In Figure 1, we selected some of the results obtained so that it is possible to understand the effect of each parameter.

The bigger the threshold, fewer bits are estimated. This can be because the algorithm will prioritize the use of the copy model rather than the fallback model. The window size results are not that linear, it cannot be too small nor too large. In our tests, the window size that obtained the best results was 10.

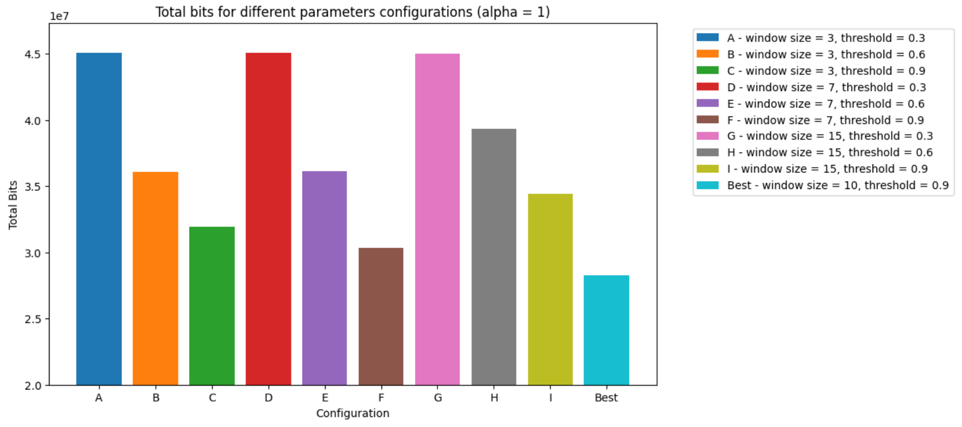

Figure 1 Comparison of total bits estimated for different parameters configurations

Model Usage

To test the how each model was being used, it was created the example file *test_fallback.txt*. We ran our cpm program with the default values (alpha = 1, window size = 10 and threshold = 0.9) and registered what model was used in each iteration. Shown in Figure 2, in the beginning, when there is not much knowledge, the fallback is the model used, but after a pattern is identified the most used model is the repeat model as expected.

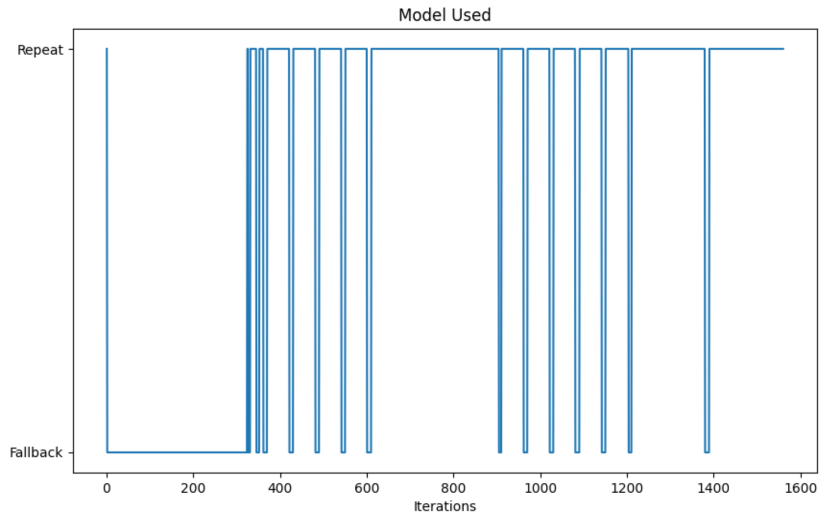

Figure 2 Model used at each iteration for test_fallback.txt file

Different Mutation Levels

To test how the randomness would affect the estimated bits needed to compress the *chry.txt* file, we used the *mutate* program to mutate the file with different mutation probabilities. As shown in Figure 3, the level of randomness in a file directly impacts its compressibility. Files with higher degree of randomness are more difficult to compress, to the point where it is almost impossible to do so.

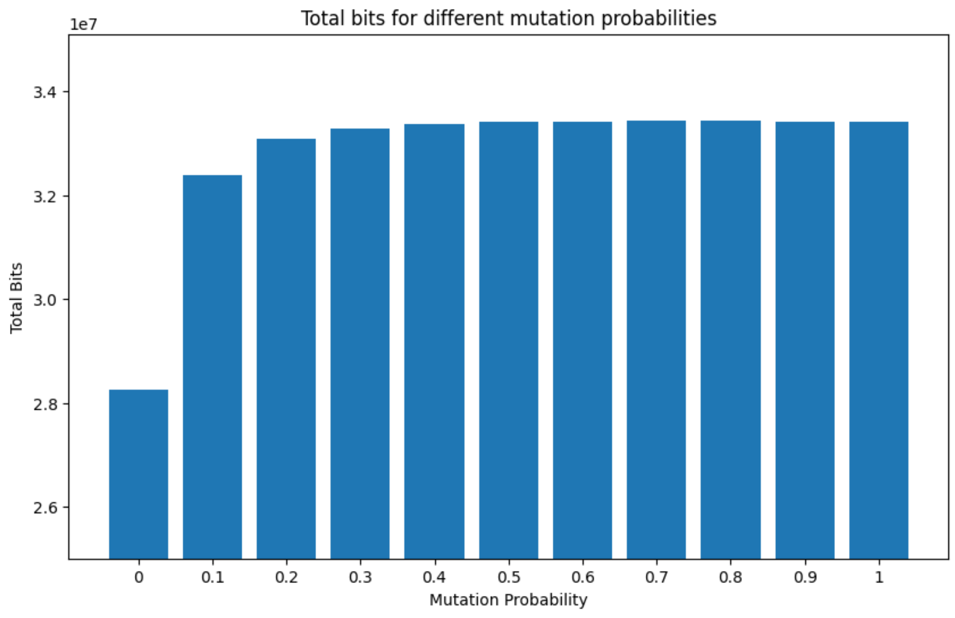

Figure 3 Comparison of total bits estimated for different mutation probabilities

Compare With Other Compressors

The Table 1 reinforces what has been previously explained that the greater the dissimilarity and randomness of the file the more bits will be required to represent it. This is evident in the table, where we observe an increase in the number of bits when using mutated files compared to the unmutated file. It is important to note that this increase occurs not only in our program but is a characteristic of compression algorithms in general.

Table 1: Comparison of file sizes for different compression algorithms

	Chry.txt	Mutate prob = 0.3	Mutate prob = 0.7
Gzip	6,2 MB (6227386 bytes)	6,6 MB (6640744 bytes)	6,6 MB (6624992 bytes)
Bzip2	5,7 MB (5692456 bytes)	6,2 MB (6203959 bytes)	6,2 MB (6197458 bytes)
Zstd	6,2 MB (6168131 bytes)	7,0 MB (7029143 bytes)	7,1 MB (7071657 bytes)
Cpm	3,5 MB (3531751 bytes)	4,1 MB (4161528 bytes)	4,1 MB (4178498 bytes)