Copy models for data compression

Lab Work 1

Bruno Nunes | 80614
Catarina Marques | 81382
David Raposo | 93395
Gonçalo Machado | 98359

Introduction

Data compression is essential for the efficient storage, transmission and processing of information. Through this process, it is possible to reduce the size of the data, optimizing the use of resources and improving the general performance of the computer systems. There are several compression methods, one of them is the copy model which predicts a symbol based on past occurrences.

This project explores the implementation of a copy model-based compression program (cpm) and a creation of a 'mutate' to introduce controlled changes to the data, using probabilities. The goal is to develop an algorithm capable of compressing data, identifying and exploring patterns and comparing the results with other compression methods (Zip, Gzip and 7z) in terms of time, memory and compression size.

The study considers different types of data, from textual documents to genetic sequences, and explores the impact of mutation probability on compression effectiveness.

Methodology and Algorithm

Copy Model

The copy model is a data compression algorithm that predicts future symbols based on previously seen symbols.

The developed program takes the following arguments:

- **k** - Size of an anchor
- **alpha** - Smoothing factor
- **threshold** - Used to check if an active copy model should be stopped
- **filename** - Name of file to be "compressed"

These predictions are based on an **anchor**, which is essentially a pointer to data already seen that has ***k*** number of characters equal to the last ***k*** characters read (these last **k** characters are referenced as a **window**) . Each prediction uses probability based on the **hits** and **misses**, which are the number of times that the algorithm guessed the next symbol correctly and incorrectly, respectively. The probability is then used to calculate the amount of bits needed to represent the information, through the formula -log2(probability).

However since when the probability is 0, the amount of bits needed is infinite, a value **alpha** is used alongside the **hits** and **misses** to calculate the probability through the formula (hits+alpha)/(hits + misses + 2*alpha).

The algorithm starts by reading the entire file to memory and extracting an alphabet consisting of all the different characters in the file and their frequency. These frequencies will be used by the **fallback model**, which happens when a copy model is not active, to calculate the probability of a character instead of the **hits** and **misses**. The fallback model is usually used at the beginning and when an **anchor** cannot be found.

After, the algorithm will go through the content of the file in memory character by character in a loop performing the following actions:

- Check if a copy model is active:
  - If true:
    - Get the reference character and calculate the probability using the **hits** and **misses**.
    - Increase the number of hits/misses depending if the reference character is equal/different to the current character, and calculate the bits needed to represent the information.
    - Add the bits to the total bits of the compression.
    - Check if the model needs to be stopped. This is done by keeping track of the last **k** **hits** and **misses** and comparing the performance of the active model to the threshold, stopping the model if the performance is below the threshold.
  - If false, the fallback model will be used, and a similar approach to the copy model will be used, but here the most frequent character will be used as the “correct” character and its probability is its frequency divided by the number of characters left to encode.
- Update the window to have the last **k** characters.
- If a copy model is not active, check if an anchor exists, in which case a copy model is started.
- Save the pair (window,position) in an unordered map.

The algorithm performs a read of the entire file before use the copy model for two reasons: to load the content of the file to memory avoiding the need to read the entire file again and to extract the alphabet and each character frequency, which are used by the fallback model that was developed.

The reason for making the fallback use the frequency of the characters in the content not compressed was because statistically, it is more likely to spend less bits encoding this information than simply encoding the character normally.

As for why the stopping of an active model uses the last k character compressions, it is largely due to stopping the model from terminating too soon. By enforcing a minimum size, we get a better sense of the accuracy of the model. An example to demonstrate this would be if the first 2 compressions where a hit and a miss, with most alphas the probability of predicting the correct character would be around 0.5, meaning that using any threshold above 0.5 would mean that the algorithm would end any copy model that did not start with 2 hits, even if those models would have had a better accuracy later on.

A fluxogram of the algorithm is available here.

Mutate

The **mutate** program is used to modify the contents of a file based on a mutation probability.

The program only the following arguments:

- **p** - Mutation probability
- **filename** - Name of file to be mutated

The program reads the file twice: firstly to extract the alphabet of the file, and secondly it goes character by character and according to the mutation probability decide if the current character should be changed or not, which if true it assumes an uniform distribution of the alphabet characters and chooses one at random to mutate the character into.

Dataset and Metrics

To experiment with the algorithm we used some samples of various types. We chose three book examples, one DNA sample converted to text file (.txt) and, of course, the assignment example “chry.txt” as the work base.

As we were working on a data compression algorithm, we needed some measures to address its efficiency. Here we found as most relevant the number of bits per char/byte (bpc), the default number of bits and compression ratio to study the compression efficiency, and the execution time.

The number of bits per char allows the evaluation of the compression efficiency per symbol. We measure this as an average of the total information bits estimated by the prediction method of the algorithm divided by the total number of characters/bytes of the file being compressed.

By searching the whole file’s alphabet we can determine the minimum number of bits required to represent a single symbol of a given alphabet and multiplying it by total number of symbols we get the minimum number of bits needed to compress the file which are here referred to as default number of bits.

Also to infer about the compression efficiency, the relation between the total number of bits compressed and original file size, which makes the compression ratio, is an important metric.

The execution time was also tested for evaluating the time efficiency, but this was only possible for our algorithm, and was not possible to get the execution time of the other compression algorithms tested.

Procedures

In order to evaluate our copy model compression based algorithm we performed a series of tests.

First, we experimented with different input parameters combinations of our algorithm in order to find the set of parameters that may provide the best performance in terms of data compression. This test was ran with the data sample given in the assignment (“chry.txt”). The different input parameter values used were the following:

K	4, 6, 8, 10, 12, 14, 16
Alpha	0.1, 1, 10 ,100
Threshold	0.3, 0.5, 0.8

Table 1: Used Parameters.

Secondly, we ran the algorithm with the optimal input parameter set on the remaining dataset.

To provide a comparison, we also ran other well-known compressors like 7z, gzip and zip on all datasets.

Finally, we chose two samples from our dataset (chry.txt and alice.txt) to execute character mutations with our mutate program using the mutation probability of 0.25, 0.5 and 0.75. With the mutated files we performed comparative compression tests between our copy model algorithm and the other used compressors.

Results and Analysis

This section will have the results regarding the project and its analysis.

Parameters combinations on "Chry.txt"

The results for the copy model tests with different parameter combinations are presented graphically on Figure 1**.**

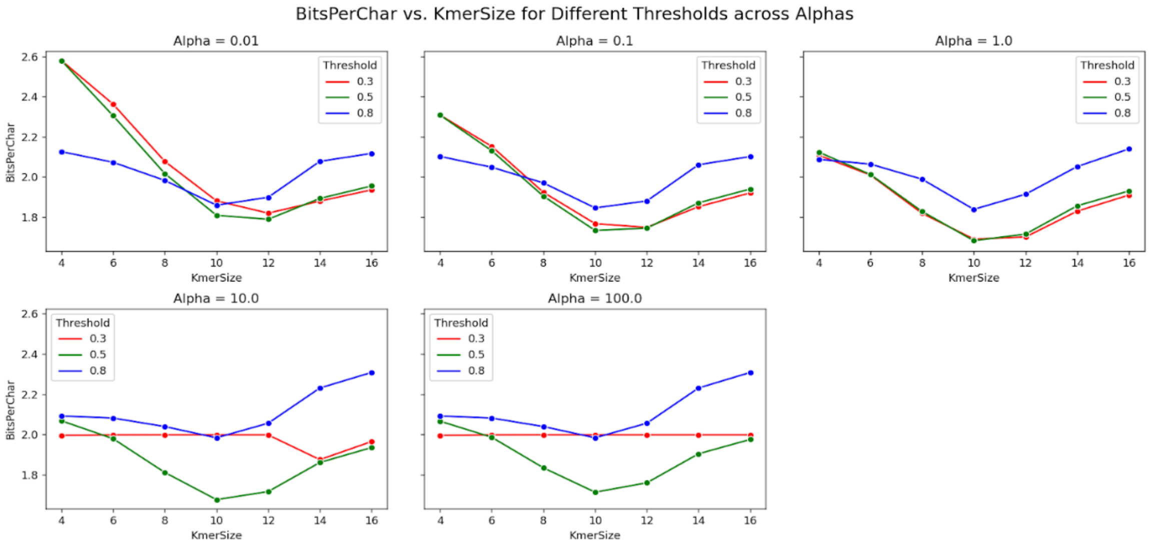


Figure 1: BitsPerChar vs KmerSize of Different Thresholds across Alphas.[Complete Results]

On a general overview it is noticeable that our copy model algorithm performed best with an anchor size of 10 to 12 characters.

By changing the smoothing factor we can observe that it has a more relevant effect when using smaller anchors on the copy model. On the other hand, for anchors of size equal or greater to the optimal, the effect of this parameter tends to vanish.

Regarding the threshold effect, the results point to the diminishing of the alpha factor effect by increasing the threshold.

Lowering the threshold for higher alpha values makes our algorithm perform closer to the default compression given by the alphabet that was calculated here as 2 bits per character, as represented on Table 3**.**

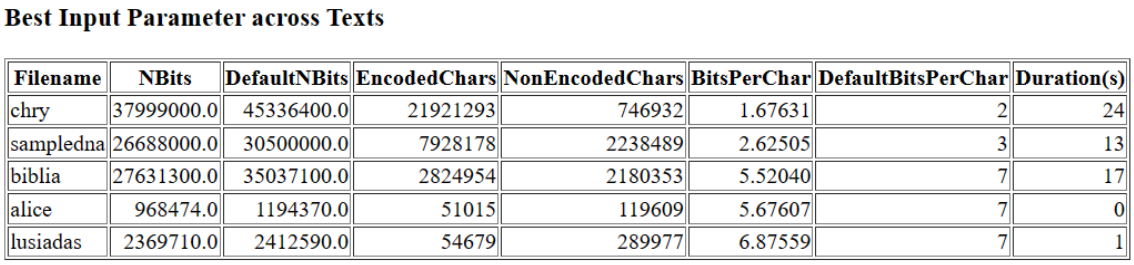
The optimal performance set of parameters appear to be reflected on the chart representative for alpha equal to 10 which are summarized on the following table.

Kmer	Alfa	Threshold
10	10	0.5

Table 2: Optimal parameter set.

Testing with Dataset

With the previous parameter set, we applied our copy model algorithm to the whole dataset described earlier. The results are presented in the following table.



Best Input Parameter across Texts

Filename	NBits	DefaultNBits	EncodedChars	NonEncodedChars	BitsPerChar	DefaultBitsPerChar	Duration(s)
chry	37999000.0	45336400.0	21921293	746932	1.67631	2	24
sampledna	26688000.0	30500000.0	7928178	2238489	2.62505	3	13
biblia	27631300.0	35037100.0	2824954	2180353	5.52040	7	17
alice	968474.0	1194370.0	51015	119609	5.67607	7	0
lusiadas	2369710.0	2412590.0	54679	289977	6.87559	7	1

Table 3: Best Input Parameter across Texts.

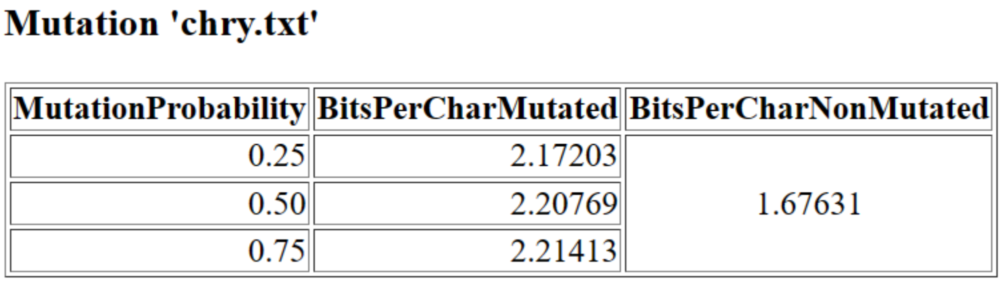
The smaller the ‘BitsPerChar’ compared to ‘DefaultBitsPerChar’, the more effective the algorithm is at compressing the data.

Analysing the table above, it can be noted that generally the ‘BitsPerChar’ is smaller than the ‘DefaultBitsPerChar’ which shows the efficiency of the copy model algorithm.

Mutated files

As referenced earlier in the report, two samples (‘alice.txt’, ‘chry.txt’) were mutated and tested with the developed copy model algorithm. The results obtained are presented on tables Table 4 and Table 5.

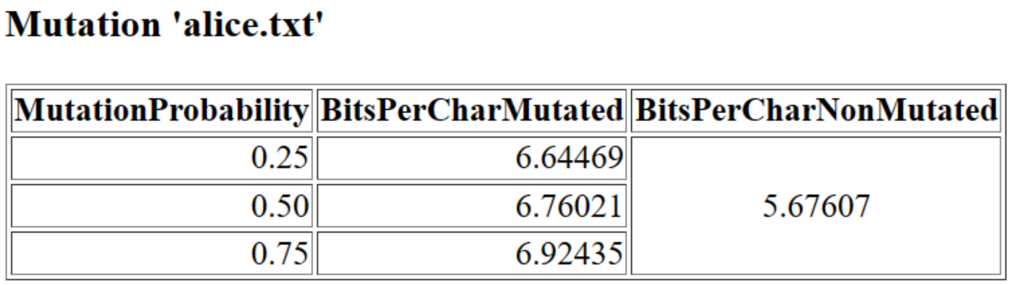
The Increase of mutation probability is followed by the increasing bits per character compressed. Additionally, it is evident that the mutation of the files tested lowers our algorithm performance.



Mutation 'chry.txt'

MutationProbability	BitsPerCharMutated	BitsPerCharNonMutated
0.25	2.17203	1.67631
0.50	2.20769	
0.75	2.21413	

Table 4: Mutation 'chry.txt'.



Mutation 'alice.txt'

MutationProbability	BitsPerCharMutated	BitsPerCharNonMutated
0.25	6.64469	5.67607
0.50	6.76021	
0.75	6.92435	

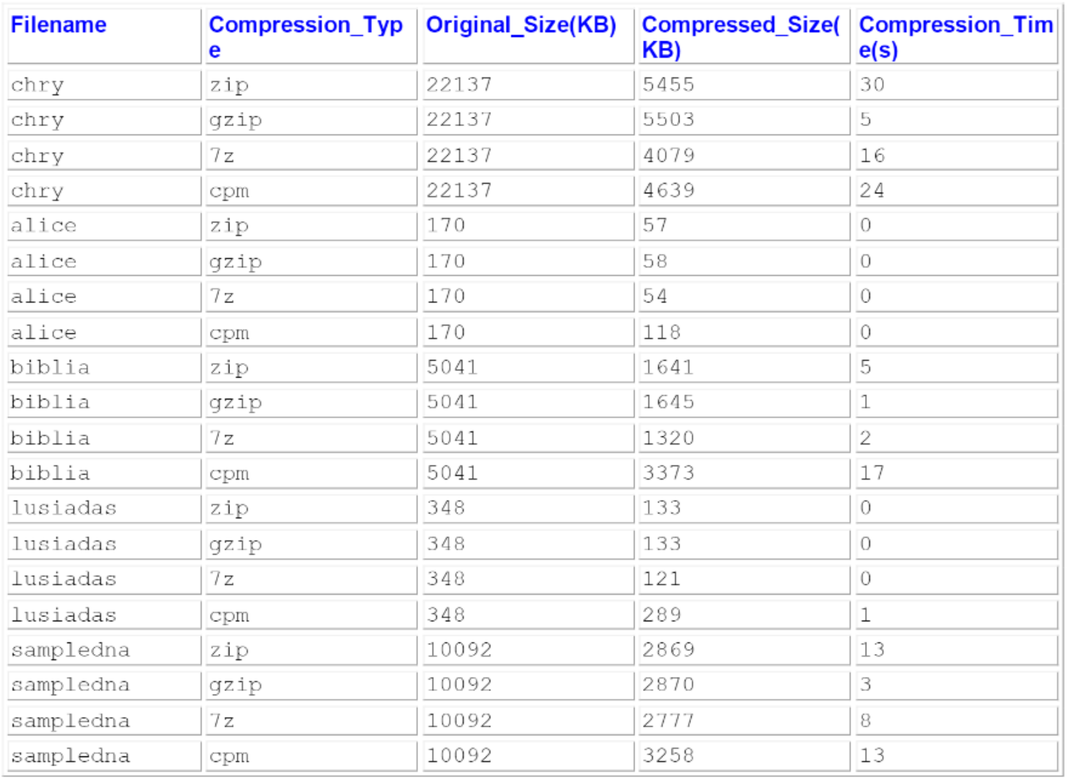
Table 5: Mutation 'alice.txt'.

Compressor Comparison

Regarding the results of compressing different files with the three selected algorithms (zip, 7z and gzip) and cpm, it can be seen in the table below that they have different performance depending on the type of data.

The zip algorithm reduced the size of the ‘chry.txt’ file from 22,137 KB to 5,455 KB, with a compression time of 30 seconds, while gzip achieved a similar compression but with a significantly shorter compression time of just 5 seconds. The 7z algorithm showed the best performance, compressing the file to 4,079 KB in just 16 seconds. Regarding the text ‘alice.txt’ all compression algorithms managed to significantly reduce the size of the original 170 KB file, with almost instantaneous compression times. In ‘lusiadas’, the 7z algorithm stood out, achieving compression to 121 KB time. The cpm compressor had the worst result, with a compressed file size of 289 KB in 1 second.

In ‘sampledna’, all compression algorithms managed to reduce the original file size by approximately 10 MB, with cpm obtaining a compressed file size of 3,258 KB in 13 seconds. It can be seen that our algorithm performs better on DNA type text files than on text files.



Filename	Compression_Type	Original_Size(KB)	Compressed_Size(KB)	Compression_Time(s)
chry	zip	22137	5455	30
chry	gzip	22137	5503	5
chry	7z	22137	4079	16
chry	cpm	22137	4639	24
alice	zip	170	57	0
alice	gzip	170	58	0
alice	7z	170	54	0
alice	cpm	170	118	0
biblia	zip	5041	1641	5
biblia	gzip	5041	1645	1
biblia	7z	5041	1320	2
biblia	cpm	5041	3373	17
lusiadas	zip	348	133	0
lusiadas	gzip	348	133	0
lusiadas	7z	348	121	0
lusiadas	cpm	348	289	1
sampledna	zip	10092	2869	13
sampledna	gzip	10092	2870	3
sampledna	7z	10092	2777	8
sampledna	cpm	10092	3258	13

Table 6: Compression result of all compressors tested on whole dataset.

Conclusion

The conclusion drawn from the developed copy model to various file types, specially the file ‘chry.txt’, offers several insights into the performance of the model.

The model showed a significant compression capability, as in the ‘chry.txt’ file there was a compression of 20% comparing with its original size. This points to the model’s high efficiency in identifying and leveraging repeating patterns in the datasets.

When applied to DNA files, the copy model had the same or better performance compared with the other compressors. This could be due to the repetitive nature of DNA sequences.

However, the copy model’s performance in compressing text files was lower than average. Text data typically contains higher diversity of patterns, which could explain this ‘poor’ performance.

Based on obtained results, the mutations showcased a negative impact on file compression, which increases the more mutations the file undergoes. This could be due to mutations introducing variability and uniqueness into the sequences, which could reduce the copy model’s ability to find repetitive patterns for compression.

Future Work

For future work, a multi anchor system would be a great choice, since instead of having just one active model, there would be several active models and the most efficient would be chosen for compression. This system could lead to an overall improvement in compression.

Leveraging competition among multiple anchors, coupled with parallel processing, could increase the compression efficiency and speed. The multi anchor system would benefit from this concurrency system.

Testing the copy model on a broader list of datasets, specially those that are larger and have more variety of content types, would provide a more comprehensive understanding of its limitations.

Finally, optimizing memory usage could be crucial as the copy model evolves to include multi anchor systems and parallel processing.