Copy models for data compression

Raquel Paradinha 102491 Paulo Pinto 103234 Miguel Matos 103341
Filipe Antão 103470

Teoria Algoritmica da Informação,
Departamento de Electrónica, Telecomunicações e Informática,
Universidade de Aveiro

March 27, 2024

Contents

Abstract

In this work we analyse and implement a lossless data compression algorithm, a Copy Model. This technique uses the premise that data sources often replicate parts of themselves over time to employ a probabilistic approach to predict future data sequences based on past occurrences.

We performed both experimental parameter tuning for our model and a comparative study against other compressors. Our solution shows a competitive performance, especially in medium to large text files, despite its computational intensity.

1 Introduction

In the digital era and with the need for instant information it becomes essential to create methods that allow to optimize storage efficiency and improve data transmission speeds.

Therefore, data compression has a fundamental role in this field, as its core objective is to identify and eliminate redundant or insignificant data elements without sacrificing essential information [3].

There are two main types of data compression, lossless and lossy. The first one is characterized by preserving all the original data, allowing for a perfect reconstruction of the original information upon decompression, and it's mainly used for executable files like documents, software applications, and texts. The other one sacrifices some level of detail once it removes unimportant information. The lossy method is commonly used in the compression of multimedia files like audio, images, and video [4].

In this work, our objective was to compress text files, so we used a lossless technique called Copy Model.

This mechanism works on the assumption that various data sources can be considered as replicating parts of themselves over time, possibly with some changes. The model compresses by predicting that the next symbol in a sequence will have already occurred in the past, as seen in Figure 1, with a given probability, and it keeps a pointer to this symbol as well as extra data to assess the accuracy of these predictions.

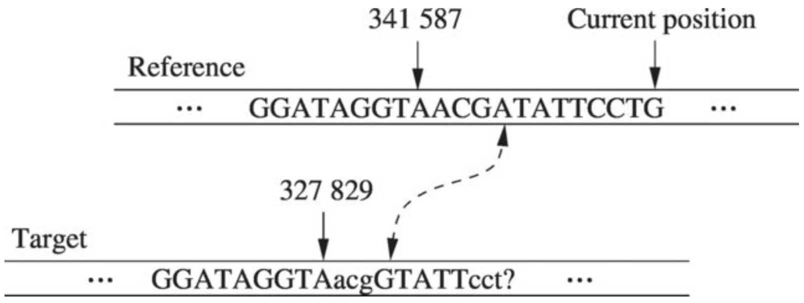


Figure 1: Copy model example. [1]

To compute these probabilities the algorithm collects counts of how often each symbol was correctly predicted (hits) versus incorrectly predicted (misses), which are then used to obtain the probability of getting and hit.

Furthermore, we've also developed a Mutate algorithm which is designed to change the content of a file at a specified probability rate.

2 Implementation

In this section, we'll detail the implementation and methodology used to develop our work.

This way, our solution is divided into two main files, the *cpm* and the *mutate*. The *cpm* makes use of two different classes, *Reader* and *CopyModel*, and is responsible for handling the command line arguments that the program can receive and initialize the needed objects.

All the classes implemented are composed of a header file, where we declare their member variables and functions, and a source file, where the actual functionality is developed.

2.1 *cpm*

This file contains the main function of the copy model program. It can receive five different command line arguments that are used to modify the performance of the model and are described in Table 3.

In the *cpm* file, we also create an instance of the class Reader and use it to read and store the contents of the file in analysis.

Then, using the stored text content and the other arguments passed to the program, we initialize a CopyModel object, which will be responsible for calculating the results of the compression performed.

Argument	**Type**	**Description**	**Values**
-f	string	Input file path	filename
-k	integer	Length of the string sequence to consider for pattern matching	[3, 15]
-t	double	Minimum prediction probability to keep using the current window	]0, 0.5]
-a	double	Smoothing parameter (α) used in probability estimation to prevent zero probability assignments	]0, 2]
-w	integer	Size of the window to use when running fallback model	[100, 500]

Table 1: Description of command line arguments for the CPM program.

2.2 *Reader* Class

The Reader class is responsible for handling the logic of accessing the input file and extracting the necessary data in a format that is suitable for the following analysis and processing.

This class has three parameters: *filePath*, *fileContent*, and *alphabet*.

The Reader also has one main method *readFile*, along with some other get methods that allow access to some of the variables stored in it.

2.2.1 readFile

The method *readFile* starts by trying to open the input file, given by the *filePath*, in binary mode and then reads one character at a time from it.

Next, the characters are appended to the string variable named *fileContent* that will be responsible for holding the entire processed text.

In addition, while reading the contents of the file, the *Reader* also stores the different letters encountered in the text in the *alphabet* variable.

2.3 *CopyModel* Class

The CopyModel class comprise the functionality of our copy model, being composed by the following member variables:

- *originalText*: the content of the input file given by the *Reader*;
- *alphabet*: the different characters encountered in the text by the *Reader*;

- *k*: the size of the strings we will compare to apply the copy model;
- *threshold*: limit value of the hit probability to change the copy model window;
- *alpha*: smothing parameter for the hit probability calculation to avoid getting a 0 value to unseen events;
- *fallbackWindowSize*: number of past characters that we will analyse to calculate the probability of the next character when there isn't an active copy model;
- *globalPointer*: position of the character we are predicting;
- *copyPointer*: past position from which we are comparing the character;
- *alphabetSize*: size of the alphabet of the text in analysis;
- *pastKStrings*: map of all the strings of size k we have already seen and the respective positions;
- *currentKString*: the string of size k we are using in a moment to look for a copy in the past;
- *totalNumberOfBits*: acumulates the number of bits needed to represent each character and is updated based on the computed probabilities.

In addition, this class also has five member functions, *run*, *findCopyModel*, *copyModel*, *fallbackModel* and *incrementGlobalPointer*, that implement the logic inherent to the copy model algorithm and are explained in detail bellow.

2.3.1 run

This method stands as the starting point of our copy model execution. It is responsible for encapsulating the two different models (copy and fallback) in a while cycle that executes until the end of the text content.

2.3.2 findCopyModel

This method is responsible for searching a match to the current k-length sequence of characters behind the current pointer in the file.

If a repetition is found, the method positions the copy pointer in the location where the copy was first found and returns the current kString to the main method. In case a copy isn't found, the function only returns an empty string.

2.3.3 copyModel

The logic for the copy model resides in this function. It starts by initializing a *Stats* object, that will be responsible for tracking the prediction success.

Then the function will enter a while loop that will run on two conditions: if the prediction success probability is *higher* than the threshold or if the number of characters predicted is still lower than the total number of characters in the original file. This function ends when one of these conditions is *false*, whether because the the model accuracy is considered too low or because we've reached the end of the file, respectively.

While these conditions are both *true* the function compares the character at *copyPointer* with the character at *globalPointer* in the original text. If they match, it means the model has successfully predicted this character, *Stats incrementHits* is called and bit cost is calculated using the formula below and added to the total sum in *totalNumberOfBits*.

$$-\log_2(\text{hit\_probability})$$

Where the hit_probability is given by the *Stats getProbability* method, described bellow.

On the other hand, when the characters don't match *Stats incrementMisses* function is called and the *totalNumberOfBits* is updated using the formula:

$$-\log_2\left(\frac{\text{comp\_prediction}}{\text{alphabetSize} - 1}\right)$$

where comp_prediction corresponds to do complementary of the hit probability, that is uniformly distributed by all the other characters of the alphabet.

2.3.4 fallbackModel

The fallbackModel consists of, as the name suggests, the fallback behavior the algorithm adopts when the copy model prediction comes to an end or while it isn't start in te beginning of the execution. Once the hit probability falls below the threshold, or in the first w iterations (with w being the fallback window size), the algorithm follows the logic determined by the fallback model.

The fallbackModel is divided into two separate behaviors. The first one is adopted for the first w iterations of the algorithm where the probability for each character follows a uniform distribution and the information is calculated based on those probabilities.

The second one is adopted when the hit probability falls below the determined threshold. First, we get the fallback string (the substring containing the w chars behind the current pointer) and calculate the counts for each character in it.

Then, using the Shannon entropy [2] the probability is calculated for each character in the substring and the information of that character is added to the total information. The formula used for these calculations is:

$$-\sum_j P(E_j)\log_2 P(E_j)$$

This fallback model runs until a new copy model is found and restarted.

2.3.5 incrementGlobalPointer

The *incrementGlobalPointer* method is called both in the copy and fallback model.

In each call to the *incrementGlobalPointer* method, the *globalPointer* variable is incremented and the substring k characters behind the global pointer, as well as the *globalPointer*, are stored in the *pastKStrings* unordered map that keeps track of the substrings and corresponding positions found.

Essentially, this function deals with the underlying logic of advancing the pointer at where we are making the predictions.

2.4 *Stats* Class

The Stats class was created to manage hits, misses, and reliability/prediction probability for each copy model, that is, for each window used. Each copy model window used is meant to use its own Stats instance since hits and misses are bound to the current window.

The only method that contains any type of logic and is not a simple setter/getter is the *getProbability* method.

2.4.1 getProbability

This method is used to obtain the probability that the next character in the current window being used by the copy model will be the same as the next character read. This probability is computed using the following formula:

$$\frac{numberOfHits + \alpha}{numberOfHits + numberOfMisses + 2\alpha}$$

A smoothing parameter (α) is used to assign a probability different than zero to all symbols in the alphabet, thus preventing assigning a probability of 0 to symbols not present in the copy window.

2.5 *mutate*

In essence, the logic within the mutate program can be split into two main parts. It first constructs the alphabet of the file by reading it and inserting all symbols into an unordered set, saving them for later use. Then, it iteratively tries to mutate each symbol - swap it with another symbol of the alphabet - of the file using the probability provided as an argument.

The following table describes the CLI arguments that the program needs to be run.

Argument	Type	Description
Input File	string	Specifies where the input file is located.
Output File	string	Specifies where the output file is located.
Mutation Probability	float	Specifies the probability to use when potentially mutating a character of the file.

Table 2: Description of command line arguments for the Mutate program.

2.5.1 constructAlphabet

This method is responsible for reading the file and returning a set containing all the different symbols that the file contains for the mutation logic. It does so by iterating through the whole file and storing the symbols within a *std::unordered_set¡char¿*.

2.5.2 mutateFile

The mutateFile method contains the main logic of the program. Using the alphabet provided by the above method and the mutation probability given as an argument, it reads the input file, potentially mutates each character read, and writes the result to the output file. The mutation decision is made using a random number generator - for each character read, if the randomly generated number is less than the mutation probability, the character is replaced with a different random character from the alphabet set. For the random number generation, the following setup was used:

- *std::random_device* was used to seed a *std::mt19937* random number generation;
- *std::uniform_real_distribution* was used to generate floating-point numbers between 0.0 and 1.0 to decide if a mutation should occur for each character
- *std::uniform_int_distribution* was used to select a random index in the alphabet vector, facilitating the selection of a new character for mutation

3 Parameter Tuning

A fundamental part of the assignment relied on the fine-tuning of the input parameters of the model, specifically the string sequence size k, the smoothing parameter $alpha$, the prediction $threshold$, and the fallback mode $windowSize$.

3.1 Setup

To analyse the behavior of the model when a given parameter changed, we tested it against the provided *chry.txt*, which contained a sequence of DNA. The interval values we used for each parameter are stated in the table below. When experimenting with a certain parameter, all others were set to a default value for all runs over the interval, as to isolate each parameter and study it separately. These values are in the *Default* column of the table.

Parameter	Start	End	Increment	Default
k	3	15	1	12
$alpha$	0.5	2	0.25	0.3
$threshold$	0.1	0.45	0.05	1
$windowSize$	150	300	25	225

Table 3: Values considered for experimentation on the *chry.txt* file.

3.2 Results

The plots present in this report are the plots in which we base the following subsections in which we analyse the behaviour change of the model when we variate one of its parameters.

3.2.1 *k*

The string sequence size significantly influences the model's performance and execution time. As k increases from 3 to 15, there's a noticeable trend towards improved compression ratios, with a reduction in the average bits per symbol. Specifically, the compression ratio improves from 3.83996 to 4.01406, and the average bits per symbol decreases from 2.08336 to 1.99299. However, this comes at the cost of increased execution time, growing from 2.925 seconds to 18.416 seconds.

Since our model checks the current k-length string against all previous ones to look for a match, the comparison operation grows more computationally expensive as *k* increases. However, the model performance tends to increase as *k* increases too, which means that looking for longer string matches improves the copy process, albeit at the cost of increased computation requirements.

3.2.2 *alpha*

This smoothing parameter is used to ensure no symbol is assigned a probability of zero, which otherwise would be problematic in instances where the current symbol does not exist in the copy window. This is made using the formula discussed before.

Adjusting alpha from 0.5 to 2 demonstrates a slight enhancement in the compression ratio (from 4.00506 to 4.01194) and a slight decrease in average bits per symbol (from 1.99747 to 1.99405). These changes indicate that higher alpha values can slightly improve compression efficiency, albeit with minimal impact on the execution time, which remains relatively constant across different alpha settings.

3.2.3 *threshold*

The higher threshold parameter effectively means that the model will allow fewer misses on the copy model mode and will resort to the fallback mode more often.

As we increase this threshold from 0.1 to 0.45, there's a marked impact on both the compression ratio and execution time. Initially, the compression ratio and average bits per symbol remain constant up to a threshold of 0.3, beyond which they start to deteriorate (from 4.01042 to 3.65953 for the compression ratio, and from 1.99481 to 2.18608 for average bits per symbol). Concurrently, the execution time increases, particularly noticeable beyond the 0.25 threshold. This indicates that higher thresholds lead to less effective compression and longer processing times, likely due to the model being more rigorous on the similarity of the current string and the copy one.

3.2.4 *windowSize*

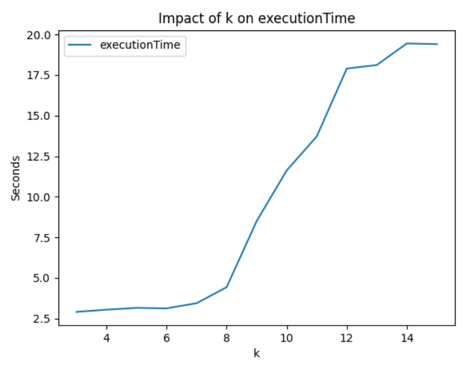
The window size affects the behavior of the fallback model, and not of the copy model. The variance in its values showed no effect on compression ratio but increased execution time. These results are most likely due to the model relying more on the copy model for the given parameters. The increase in execution time is due to the fact that the fallback model computes frequencies of symbols inside the fallback window - if it is bigger, more operations are executed.

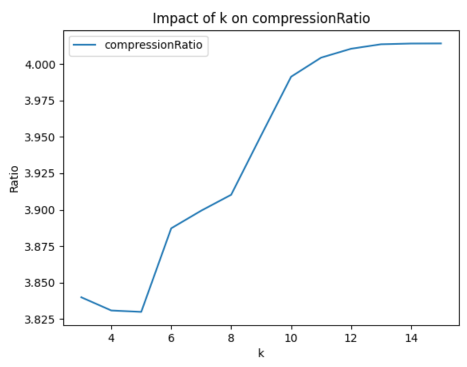
3.2.5 Other notes

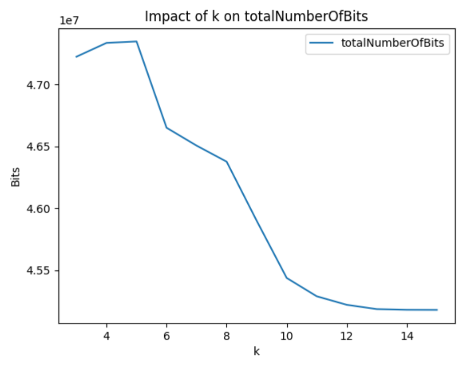
Its is worth noting a somewhat interesting relationship between the *alpha* and the *threshold* parameters. As stated in this project's syllabus, a value of *alpha* equal to 1 means that the initial prediction of our copy models are always 0.5, since there are still no hits nor misses. This means that if our *threshold* is higher than this initial correct prediction probability, which by itself is conditioned by the *alpha* parameter, the model will never be able to use the copy mode, since it deems all models too inaccurate before even checking the first symbol.

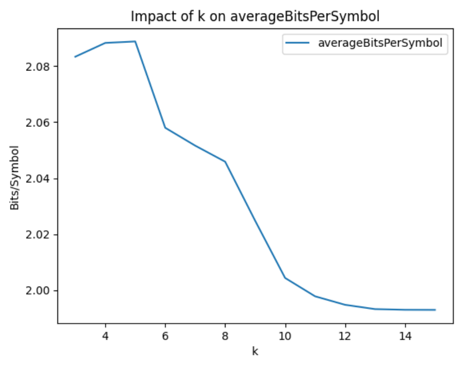
4 Comparisons

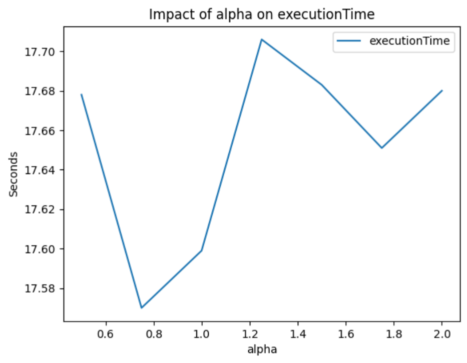
To ensure the validity of our copy model, some comparisons with other compression algorithms were performed. The compression algorithms being compared are the Gzip, Bzip2, Zstd, Xz and Lz4. Metrics such as the Compression Ratio, the Average number of Bits per Symbol, the Compressed File size and the time taken to perform the compression were analyzed and compared. Three text files were used to perform the tests on: chry.txt, the file provided in elearning (20MB); the random_file.txt, a large file (90MB) generated by us; and the 1984_excerpt.txt, which is a relatively small file (5Kb). The results can be seen in Table 4.

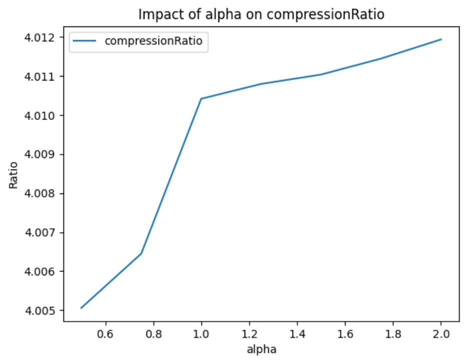

Impact of k on executionTime
executionTime
Seconds
k
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
4
6
8
10
12
14

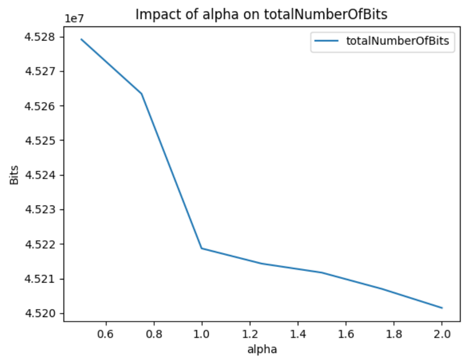

Impact of k on compressionRatio
compressionRatio
Ratio
k
4.000
3.975
3.950
3.925
3.900
3.875
3.850
3.825
4
6
8
10
12
14

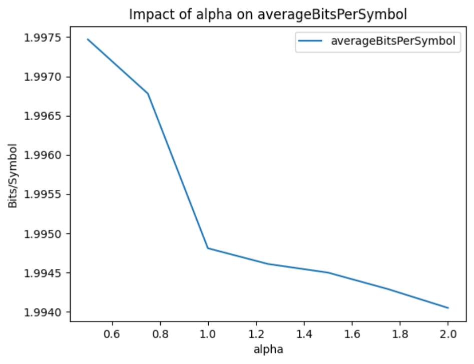

Impact of k on totalNumberOfBits
1e7
totalNumberOfBits
Bits
k
4.70
4.65
4.60
4.55
4
6
8
10
12
14

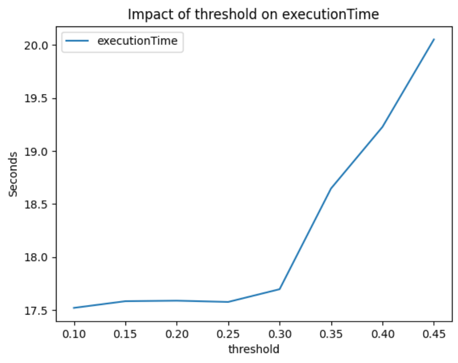

Impact of k on averageBitsPerSymbol
averageBitsPerSymbol
Bits/Symbol
k
2.08
2.06
2.04
2.02
2.00
4
6
8
10
12
14

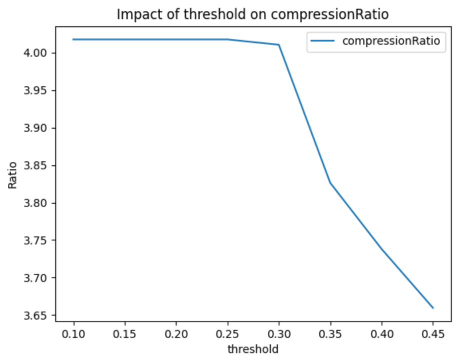

Impact of alpha on executionTime
executionTime
Seconds
alpha
17.70
17.68
17.66
17.64
17.62
17.60
17.58
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0

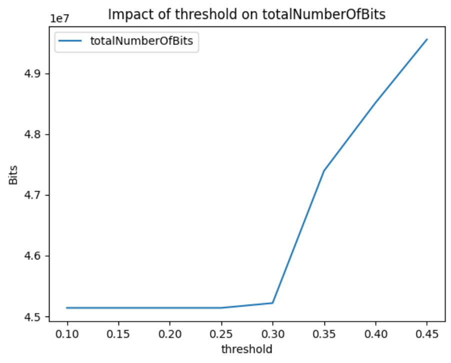

Impact of alpha on compressionRatio
compressionRatio
Ratio
alpha
4.012
4.011
4.010
4.009
4.008
4.007
4.006
4.005
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0

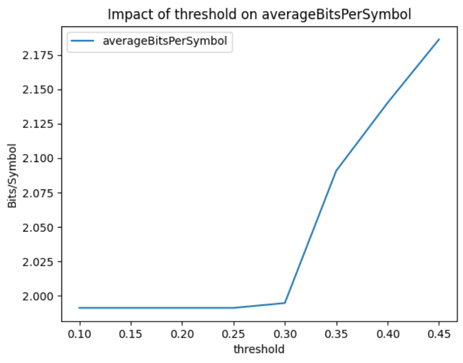

Impact of alpha on totalNumberOfBits
1e7
totalNumberOfBits
Bits
alpha
4.528
4.527
4.526
4.525
4.524
4.523
4.522
4.521
4.520
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0

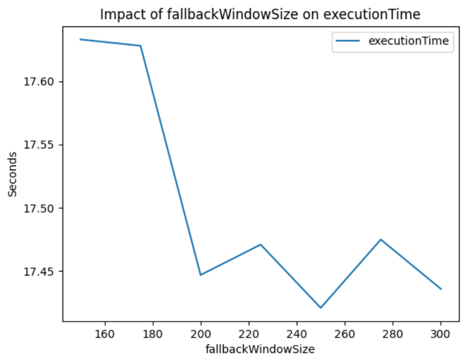

Impact of alpha on averageBitsPerSymbol
averageBitsPerSymbol
Bits/Symbol
alpha
1.9975
1.9970
1.9965
1.9960
1.9955
1.9950
1.9945
1.9940
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0

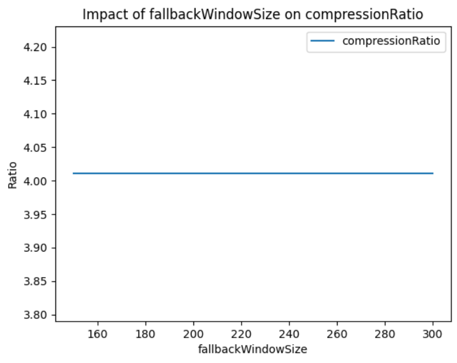

Impact of threshold on executionTime
executionTime
Seconds
threshold
20.0
19.5
19.0
18.5
18.0
17.5
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45

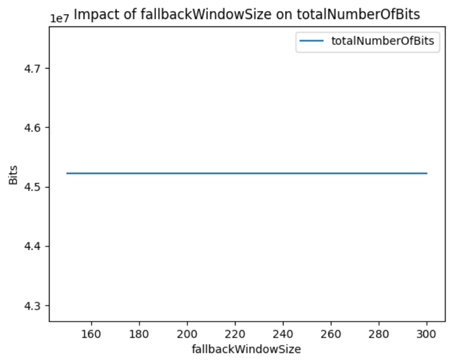

Impact of threshold on compressionRatio
compressionRatio
Ratio
threshold
4.00
3.95
3.90
3.85
3.80
3.75
3.70
3.65
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45

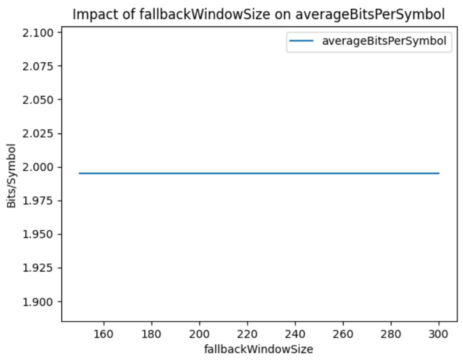

Impact of threshold on totalNumberOfBits
1e7
totalNumberOfBits
Bits
threshold
4.9
4.8
4.7
4.6
4.5
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45


Impact of threshold on averageBitsPerSymbol
averageBitsPerSymbol
Bits/Symbol
threshold
2.175
2.150
2.125
2.100
2.075
2.050
2.025
2.000
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45


Impact of fallbackWindowSize on executionTime
executionTime
Seconds
fallbackWindowSize
17.60
17.55
17.50
17.45
160
180
200
220
240
260
280
300


Impact of fallbackWindowSize on compressionRatio
compressionRatio
Ratio
fallbackWindowSize
4.20
4.15
4.10
4.05
4.00
3.95
3.90
3.85
3.80
160
180
200
220
240
260
280
300


Impact of fallbackWindowSize on totalNumberOfBits
1e7
totalNumberOfBits
Bits
fallbackWindowSize
4.7
4.6
4.5
4.4
4.3
160
180
200
220
240
260
280
300


Impact of fallbackWindowSize on averageBitsPerSymbol
averageBitsPerSymbol
Bits/Symbol
fallbackWindowSize
2.100
2.075
2.050
2.025
2.000
1.975
1.950
1.925
1.900
160
180
200
220
240
260
280
300

Algorithm	File	Comp. Ratio	Avg Bits/Symbol	Time (s)	Compressed Size (bits)
gzip	chry.txt	3.64	2.20	5.61	49,819,088
bzip2	chry.txt	3.98	2.01	2.20	45,539,648
xz	chry.txt	5.34	1.50	25.76	33,929,376
lz4	chry.txt	1.85	4.32	0.09	97,830,704
zstd	chry.txt	3.72	2.15	0.13	48,771,624
Our Model	chry.txt	4.01	2.00	32.68	45,183,900
gzip	random_file.txt	1.57	5.08	4.01	508,220,800
bzip2	random_file.txt	1.67	4.78	8.93	478,325,632
xz	random_file.txt	1.65	4.86	26.03	485,751,104
lz4	random_file.txt	1.00	8.00	0.08	800,000,888
zstd	random_file.txt	1.67	4.78	0.43	478,113,616
Our Model	random_file.txt	1.51	5.29	366.32	529,261,000
gzip	1984_excerpt.txt	2.04	4.91	0.009	21,704
bzip2	1984_excerpt.txt	2.14	4.68	0.010	20,688
xz	1984_excerpt.txt	2.00	5.00	0.011	22,144
lz4	1984_excerpt.txt	1.33	7.52	0.008	33,280
zstd	1984_excerpt.txt	2.01	4.99	0.009	22,104
Our Model	1984_excerpt.txt	1.51	4.24	0.011	23,509.3

Table 4: Compression Performance Comparison

After conducting the tests, and after analyzing the results, we can observe that our copy model has a good performance, particularly when compressing medium and large text files. This can be seen by the above average compression ratios, which are only topped in all tests by the Xz algorithm. However, it's important to note that our copy model has some drawbacks, in particular its high execution time.

5 Problems

The approach presented and tested in this work is not the only one we tried to implement. Since the current approach only compares the symbols to the first pointer found, we wanted to implement one that would compare the symbols to all the pointers found, and then choose the best pointer (the one whose probability was the last to fall below the threshold). Additionally, to increase the performance, we also resorted to multi threading, creating a thread for each pointer and its prediction process.

We implemented this logic but, for some reason we couldn't figure out in time, the results returned didn't match the theoretical boost in performance this approach would give us. So we decided to discard it. Nonetheless, all the code and its implementation is in the branch threads in the GitHub repository.

6 Conclusion

In this work, we achieved all our predetermined goals. A copy model was implemented and, despite the time inefficiency, still performed well and returned some good results when compared to other compression algorithms. As for the mutate program, it was straightforward to implement.

Unfortunately, we couldn't conclude the correct implementation of the copy model using multi threading, which would give us an algorithm with better performance.

References

[1] Diogo Pratas Armando J. Pinho and Sara P. Garcia. *GReEn: a tool for efficient compression of genomeresequencing data*. `https://sweet.ua.pt/pratas/papers/Pinho-2011a.pdf`. Accessed: 2024-03-26. 2011.

[2] Armando J. Pinho. *Some notes for the course ALGORITHMIC INFORMATION THEORY*. Accessed: 2024-03-16. 2023.

[3] Timescale. *What Is Data Compression and How Does It Work?* URL: `https://www.timescale.com/learn/what-is-data-compression-and-how-does-it-work` (visited on 03/26/2024).

[4] Kashyap Vyas. *Data Compression Techniques*. URL: `https://www.datamation.com/big-data/data-compression/` (visited on 03/26/2024).