



## *Copy Models for Data Compression*

### **Mestrado em Engenharia Informática**

Universidade de Aveiro

DETI - Aveiro, Portugal

UC 40752 - Teoria Algorítmica da Informação  
2022/2023

José Trigo 98597  
Pedro Monteiro 97484  
Eduardo Fernandes 98512

# Copy Models for Data Compression

March 30, 2023

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Objectives</b>	<b>4</b>
<b>3</b>	<b>Implementation</b>	<b>5</b>
3.1	Organization . . . . .	5
3.2	Developed programs . . . . .	5
3.3	Copy Model ( <i>cpm</i> ) . . . . .	5
3.3.1	Data structures . . . . .	5
3.3.2	Strategy addressed . . . . .	5
3.3.3	Read the text file . . . . .	6
3.3.4	Store the k-string's in the data structures . . . . .	6
3.3.5	Get the predicted character and modify the counters . . . . .	6
3.3.6	Calculate the probabilities . . . . .	7
3.4	Automatic Text Generator ( <i>cpm_gen</i> ) . . . . .	7
3.4.1	Data structures . . . . .	7
3.4.2	Strategy addressed . . . . .	7
3.4.3	Read the text file . . . . .	7
3.4.4	Load the <i>char_counting</i> map structure . . . . .	8
3.4.5	Generate the output sequence . . . . .	8
3.4.6	Results . . . . .	9
<b>4</b>	<b>Tested Approaches</b>	<b>9</b>
<b>5</b>	<b>Results</b>	<b>10</b>
<b>6</b>	<b>Conclusion</b>	<b>12</b>

# 1 Introduction

Algorithmic Information Theory (AIT) is a field of study that focuses on the representation, communication, and processing of information in efficient and concise ways. Data compression, a cornerstone of AIT, aims to minimize the size of data while preserving its original structure and meaning. This reduction in size is crucial for efficient storage, transmission, and processing of data in various applications such as multimedia, data communications, and data mining. One promising approach to achieve data compression is through the use of copy models.

Copy models for data compression capitalize on the inherent self-similarities within data sources. By identifying and exploiting these similarities, we can efficiently represent data through references to previously occurring parts with minor modifications. This report delves into the intricacies of copy models, exploring how they predict the next outcome in a data sequence by maintaining a pointer to the symbol being replicated and estimating the reliability of such predictions.

We will discuss the estimation of probabilities involved in this process, addressing the challenge of assigning probabilities to previously unseen events using a smoothing parameter. The report will also touch upon how these probabilities can be utilized to estimate the amount of information required to represent a new symbol, ultimately driving the effectiveness of data compression.

Furthermore, we will introduce and examine a program for automatic text generation, based on the copy model, named "cpm gen." This program, leveraging the principles of copy models, will be capable of generating text that follows a given example text's structure and style.

Throughout the course of this report, we aim to provide an in-depth understanding of copy models in the context of algorithmic information theory and their application in both data compression and automatic text generation.

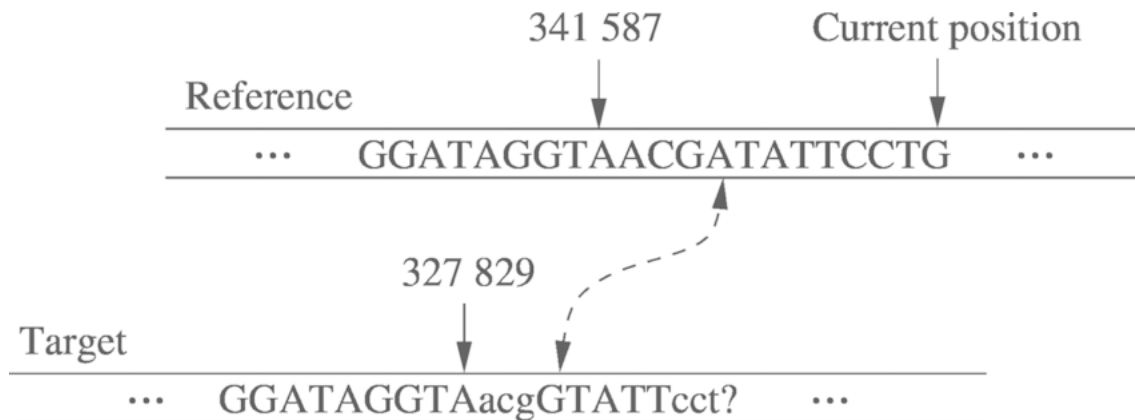


Figure 1: Copy Model Example.

## 2 Objectives

The main objective of this assignment is to develop a program that predicts the next characters in a DNA sequence by utilizing sub-sequences of size  $k$  (where  $k > 0$ ) that have been previously identified.

Additionally, a generator was created based on the previous program, which takes a character sequence file as input for automatic text generation.

Throughout this report, we will explore the data structures and algorithms used in the implementation of these programs, and present the results obtained from testing the programs using different files and values of  $k$  and  $\alpha$ .

## 3 Implementation

In this section, we will address how we perform our implementation as well as organize the code and exercise files. The functions, structures used, and the strategies applied for each of the algorithms will also be discussed.

### 3.1 Organization

The image below shows the structure of the project. In the **src** folder are all the files related to the developed code, for the Copy Model, *cpm*, and for the Automatic Text Generator, *cpm\_gen*. In the **examples** folder it is possible to find several text files that were used to carry out tests. Furthermore, **README.md** describes how to run the program and finally, in the **report** folder there is this report that describes what was done during the development of the entire project.

### 3.2 Developed programs

For this project, as previously mentioned, two programs were developed. Each one refers to the topic of copy models. The first one, *cpm.cpp*, implements a copy model. This program estimates the total number of bits needed to encode a certain file, as well as the average number of bits per symbol.

The second and the last one, *cpm\_gen.cpp* is a program for automatic text generation, that based on an example text, is able to generate text that follows that model.

In the next sections we will address each of the problems separately.

### 3.3 Copy Model (*cpm*)

#### 3.3.1 Data structures

To achieve the goals of the copy model we have implemented some structures:

- *chunks\_positions*: map of k-strings to their positions in the file;
- *chunks\_vector*: vector of chunks of length k;
- *positions\_count*: map containing the number of fails for each position;

#### 3.3.2 Strategy addressed

The following list explicit the strategy addressed in the copy model:

1. Get the inline command arguments for the file, the size of the k-string, the value of *alpha* and the fail threshold (file, k, alpha, fail\_threshold respectively);
2. Read the file and assign to a string;
3. Iterate the data from file and add the current k-string to the *chunks\_vector* as well as the position in the *chunks\_positions* map;
4. Get the predicted character by getting the first position in the vector of positions for that chunk, going to that position in the *chunks\_vector* and retrieving the character immediately after the k-string;
5. Verify if its a hit or fail:

- If its a hit, increment the hits counter;
  - If its a fail, increment the fails counter and deletes the current position if the fail counter is equal or bigger than *fail\_threshold*;
6. Lastly, calculate the probability of a correct prediction and update the estimated total number of bits.

One thing to notice is that this model only starts to increase the hits/fails counters when the first repeated k-string appears.

### 3.3.3 Read the text file

In order to read the text file we created a function called *read\_file()* that reads the text in the file and store it in a string.

### 3.3.4 Store the k-string's in the data structures

This logic is done inside the *main* function of the program. It iterates each k-string and adds each one to the map with the current position. If the chunk is already in the map, only adds the position to the vector associated.

Also stores the chunks in the *chunks\_vector* so that, in the future, it can get the next character of a chunk for a certain position.

The code for the section addressed is below.

```
for (int i = 0; i < data.size() - k; ++i){

    // extract k length chunk from the file and add it to the vector of chunks
    std::string current_chunk = data.substr(i, k);
    chunks_vector.push_back(current_chunk);
    //cout << "Current chunk: " << current_chunk << endl;

    // if the current chunk is not in the map add it to the map
    if (chunk_positions.find(current_chunk) == chunk_positions.end())
    {
        chunk_positions[current_chunk] = std::vector<int>{i};
        continue;
    }

    // add the position of the current chunk to the map
    chunk_positions[current_chunk].push_back(i);

    // if this is the last chunk, break out of the loop
    if (i + k >= data.size())
        break;
    .
    .
    .
}
```

### 3.3.5 Get the predicted character and modify the counters

As it was already mentioned before, we get the predicted character by getting the first position in the vector of positions for that chunk, going to that position in the *chunks\_vector* and retrieving the

character immediately after the chunk.

To get the actual character, we go to the position of the current chunk in the full text string, returning the character immediately after the current k-string.

After getting this two characters, we compare them in order to know if we should increase the hits counter or the fails counter.

In order to improve the results, we have implemented a *fail\_threshold* which restricts the number of allowed failures at each position. Specifically, if a position associated with a certain k-string has a number of failures equal to or exceeding the *fail\_threshold*, that position is removed and replaced by another position which becomes the first one in the vector of positions for that k-string.

### 3.3.6 Calculate the probabilities

The calculation of the probabilities happens every time the hits/fails counters increase. The formulas that we used are the ones provided by the teachers in assignment.

```
// calculate the probability of a correct prediction and
// update the estimated total number of bits
prob = (hits + alpha) / (hits + fails + 2 * alpha);
estimated_bits += -log2(prob);
```

## 3.4 Automatic Text Generator (*cpm\_gen*)

### 3.4.1 Data structures

As *cpm\_gen* is based on the code of the copy model from exercise 1, it maintains some of its data structures, including a 2D array that stores the k-strings the model pointer is pointing to. A structure that was only implemented in this program is a map, *char\_counting*, that uses a string (k-string) as a key and another map as a value, which will contain each character following the k-string along with the number of times it appears after the specific string.

### 3.4.2 Strategy addressed

The following list explicit the strategy addressed in this exercise:

1. Get the inline command arguments for the file, the size of the k-string and the desired output text length (file, k, text\_length, respectively);
2. Read the file and assign to a string;
3. Iterate the data from file and count the frequency of the next character for each k-string;
4. Generate the output sequence using a random value between 0 and 1 to determine which character to add after each chunk of text.

Topics 2, 3 and 4 will be covered in more detail below.

### 3.4.3 Read the text file

The function *read\_file()* aims to read the text file in order to obtain the DNA sequence in a string format. This is done using the same function from the *cpm*.

### 3.4.4 Load the *char\_counting* map structure

For this purpose, the *load\_map()* function is called, which has as arguments the map where the counters of each character for each k-string will be stored, and the vector where each k-string will be added. This is shown in the code snippet below.

```
// For each chunk of text, add the counters of characters to a map
void load_map(int k, string &data, std::map<std::string,
std::map<char, int>> &char_counting, std::vector<std::string> &chunks){

    // Iterate through the data, creating k-length chunks
    for (int i = 0; i < data.size() - k; ++i)
    {
        // Extract a k-length chunk from the data
        std::string current_chunk = data.substr(i, k);

        // Add the current chunk to the chunks vector
        chunks.push_back(current_chunk);

        // Get the character immediately following the current chunk
        char next_char = data[(i+k)];

        // If the current chunk is not yet in the char_counting map,
        // add it with the next character count initialized to 1
        if (char_counting.find(current_chunk) == char_counting.end()) {
            char_counting[current_chunk][next_char] = 1;
            continue;
        }

        // If the current chunk is already in the char_counting map,
        // increment the count for the next character
        char_counting[current_chunk][next_char] += 1;
    }
}
```

### 3.4.5 Generate the output sequence

This section of the program contains two functions:

1. *get\_next\_char(char\_count)*:
2. *generate\_text(char\_counting, text\_length, starting\_text)*:

For context, the *char\_count* is the map containing the counters of the characters for each k-string and the *starting\_text* is the k-string from where we are generating the next character.

The first function generates the next character to be added to the final text given a k-string. This is done by generating a random value between 0 and 1, which is used to determine the character. The probabilities of each character for each k-string are calculated by dividing each character's counter by the total value of all characters.

The *generate\_text* function just appends the new character to the output sequence. To achieve this, the function identifies the final k-string and subsequently calls the function above which generates the character to be appended.



### 3.4.6 Results

The following snippet is a possible way to run the program and, given the chry.txt file, generate a sequence of characters for a k equal to 5.

```
./cpm_gen chry.txt 5 1000  
GAATTC...TAAGAGGAACAATTGTAGTATT
```

The same happens here but using another text file, and generating a sequence with 10 characters taking a k-string of size 3.

```
./cpm_gen test.txt 3 10  
CTTCCAGGGA
```

A problem with this implementation is the fact that if any k-string in the text being generated is not in the map containing the counters for each chunk, the algorithm cannot determine what is the next character for that k-string.

To prevent the program from entering an infinite loop where it is constantly looking for the k-string in the map, we added another logic that verifies if the current chunk is in the map, and if it doesn't, returns the next that was generated until there. The side effect is that it does not generate a sequence with the desired length, but at least gives one that follows the example text. This may depend on the value of k as well as the sample text passed as an argument.

## 4 Tested Approaches

In the initial attempt to solve the problem, the primary objective was to predict the most frequent next character in a given sequence of text, based on the previous k characters. The approach utilized a sliding window of size k to form chunks of text, and a hashmap was created to store the positions of each chunk in the input data. To predict the next character, a function called `predict_char` was used, which took the positions of a chunk and a vector of chunks as input. The function then initialized a counter for each character and iterated through the positions to find the most frequent character. We retrieved all positions of the given chunk, iterating through them individually to identify the most common character although the process of checking all positions was extremely time-consuming.

However, in the final delivered approach, the focus was not only on predicting the next character but also on calculating the probability of correct predictions and estimating the total number of bits for the given text. In this approach, command-line arguments were used to provide input parameters, such as the filename, sliding window size k, a smoothing parameter alpha, and a fail threshold. The implementation of the `predict_char` function was simplified by directly returning the last character of the chunk immediately following the first position of the chunk.

In addition to the changes in the `predict_char` function, several other enhancements were made in the final approach. For instance, a map called `positions_count` was introduced to keep track of consecutive prediction failures for each chunk. If the number of failures for a specific chunk reached the fail threshold, the chunk would be removed from the hashmap.

Furthermore, the probability of a correct prediction was calculated using the smoothing parameter alpha. The estimated total number of bits was updated by adding the negative logarithm (base 2) of the probability of a correct prediction. Finally, the average number of bits per symbol was computed by dividing the estimated total bits by the total number of symbols in the input data.

In summary, the delivered approach not only improved the predictability of the next character but also provided insights into the probability of correct predictions and the estimated total number of bits. This was achieved through various enhancements, such as a simplified `predict_char` function, the introduction of a map to track consecutive prediction failures, and the use of a smoothing parameter to calculate the probability of correct predictions.

## 5 Results

For testing purposes, three different files were used. The same computer was used for all tests, consisting of an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz with 6 cores and 1 socket. The results presented were also obtained by taking the average value of several tests.

The **chry.txt** file, provided by the course's teachers. The **gen.txt** file that was previously generated using the generator developed in exercise 2. And, even a smaller file, **test.txt**, for model debugging purposes. All these test files can be found in the examples folder.

As for the **test.txt** file, it only contains a 13-character sequence A.

Considering  $k=4$  and  $\alpha=1$ , the following results were obtained:

- Hits: 8
- Fails: 0
- Total Bits: 2.32193
- Average Bits per Symbol: 0.17861

Also, considering the formula for calculating the hit probability provided in the statement,

$$\frac{nhits+\alpha}{nHits+nfails+2*\alpha}$$

A hit probability of 0.9 was obtained for the last character. This means that there is a 90% chance that the character following the sequence will be an A. This probability can be verified by substituting the formula parameters with the obtained value:

$$\frac{8+1}{8+0+2*1} = \frac{9}{10}$$

This high probability is due to the fact that for this test/debug file there are only equal characters. Regarding the test file **gen.txt** that has about 5MB and was generated based on the **chry.txt** file available in elearning, using the exercise 2 generator, tests were carried out for different values for K and for the estimated total number of bits.

For the graph shown below, a fail threshold equal to 3 and  $\alpha$  equal to 1 was considered.

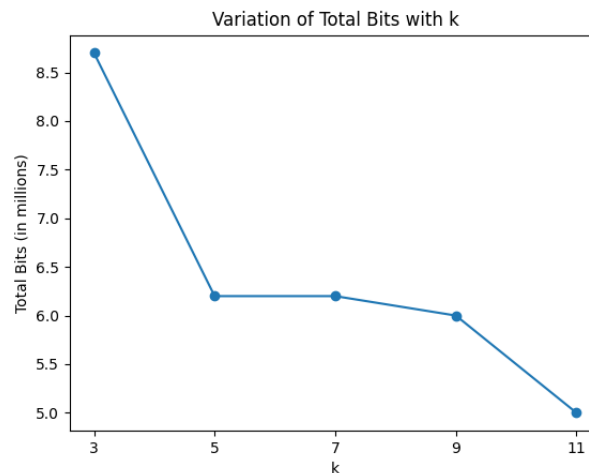


Figure 2: Total Bits Variation with K values.

As can be seen, for the same values of  $\alpha$  and threshold, the value chosen for k, that is, the size of

the characters sequence, has a high impact on the file compression, with greater compression for higher values of  $k$ .

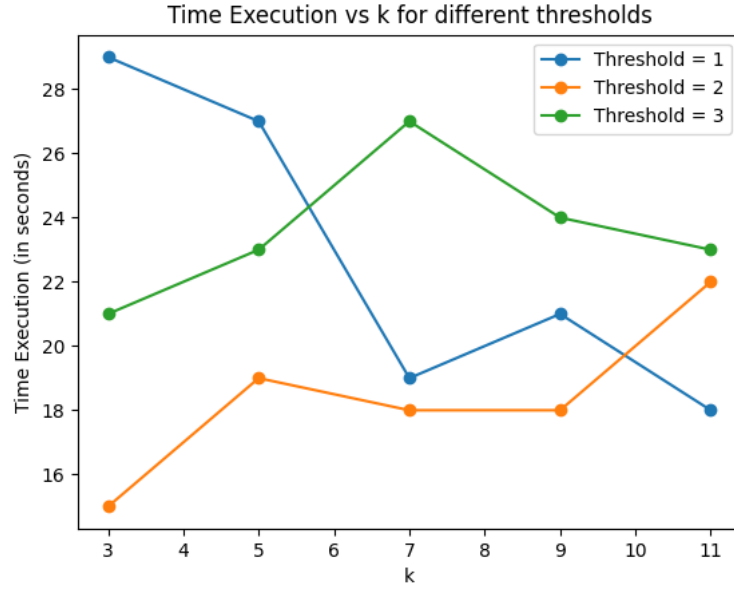


Figure 3: Variation of Time Execution with K and Threshold.

Based on the chart above, we can see that the time execution of the model varies with the value of  $k$  and the threshold parameter used.

Overall, we can observe that the time execution decreases as the value of  $k$  increases, for all threshold values. This suggests that the algorithm becomes faster as it is allowed to have more fails for a certain position.

We can also see that the time execution varies significantly between different threshold values, particularly for smaller values of  $k$ . For example, for  $k=3$ , the time execution for threshold=1 is around 29 seconds, while the time execution for threshold=3 is around 19 seconds. This suggests that the choice of threshold can have a significant impact on the performance of the algorithm, particularly for smaller values of  $k$ .

Overall, we can conclude that the performance of the algorithm depends on both  $k$  and threshold values, and that the choice of these parameters should be carefully considered to achieve optimal performance. Additionally, increasing the value of  $k$  can generally improve the performance of the algorithm.

In the case of the **chry.txt** file, which is about 22MB, several results were recorded in the table below, taking into account different values for  $k$  and for the threshold, always keeping the alpha value equal to 1.

The table below shows that as the value of  $k$  increases, the estimated total bits decreases, and the average bits per symbol also decreases. This suggests that higher values of  $k$  lead to better compression. Additionally, increasing the threshold value also seems to improve compression.

However, it is worth noting that the real time taken for the experiments to run is not consistent across the different parameter settings. For instance, increasing the value of  $k$  from 7 to 9 led to a significant increase in the time taken for the experiment to run. Therefore, there may be a trade-off between the quality of compression and the time taken to achieve it.

Overall, the data provided suggest that the parameters  $k$ , alpha, and threshold can have a significant impact on the quality of compression achieved, and that finding the optimal values of these parameters may require balancing trade-offs between compression quality and computational efficiency. Note the fact that in all tests the same value was always used for alpha,  $\alpha = 1$ .

<b>k</b>	<b>Threshold</b>	<b>Estimated Total Bits (millions)</b>	<b>Avg. Bits per Symbol</b>	<b>Time (seconds)</b>
5	3	39.2	1.73	96
5	5	39.3	1.74	91
7	3	37.4	1.65	84
7	5	37.5	1.65	105
9	3	33.6	1.48	126
9	5	33.9	1.50	119
11	3	24.9	1.10	137
11	5	25.3	1.12	140

Table 1: Experimental Results with Varying Values of k and Threshold

## 6 Conclusion

In conclusion, this report explored the use of copy models for data compression and automatic text generation. The report discussed how copy models can identify and exploit self-similarities within data sources, efficiently representing data through references to previously occurring parts with minor modifications. The estimation of probabilities involved in the copy model process was addressed, including the challenge of assigning probabilities to previously unseen events using a smoothing parameter.

Two programs were created, one for implementing the copy model and the other for generating text automatically. In this report, we focused on the implementation of the automatic text generator, discussing the data structures utilized, as well as the algorithms' strategies. Additionally, we presented the results of our implementation, including various tests conducted with different files and different values of k and alpha.

Overall, this report provided an in-depth understanding of copy models in the context of algorithmic information theory and their application in both data compression and automatic text generation. The developed programs can be useful tools for various applications, such as data storage, transmission, and processing, and can potentially lead to further research in the field.