Assignment 1: Report

Copy Model for Data Compression

27th of March 2024

Authors

102534 Rafael Gonçalves

103154 João Fonseca

103183 Diogo Paiva

Professors

Prof. Dr. Armando J. Pinho

Prof. Diogo Pratas

Algorithmic Theory of Information - 2023/24

MSc in Informatics Engineering

University of Aveiro

Table of contents

Table of figures

1 Introduction

1.1 Purpose of the report

The purpose of the report is to document the work developed by our group, for the first assignment of the Algorithmic Theory of Information course (2023/24), where we were proposed the exploration of the usage of copy models for data compression. [1]

1.2 Data compression using copy models

Data compression is obtained through the exploration of self-similarities. Several types of data have specific structural properties that can be taken advantage of by different compression techniques. For instance, when compressing a text file written in English, one can consider the average occurrence of characters in the English language to encode more frequent letters with a lesser amount of information (bits).

In this report, we explore compression of data using copy models. This approach relies on the idea that some data sources can be viewed as being reproduced by replicating parts already produced in the past, with some possible modifications, like genome resequencing data [2]. A copy model predicts that the next outcome will be a symbol that has occurred in the past.

To estimate probabilities, we consider the number of hits N_h (symbols predicted correctly), the number of misses N_m (symbols predicted wrongly) and a smoothing factor α to prevent assigning a zero probability to events not seen during the construction of the model. The value "2" refers to the number of events being predicted (hit or miss).

$$P(hit) \approx \frac{N_h + \alpha}{N_h + N_h + 2\alpha}$$

After computing the copy model, we can estimate the amount of information that a new symbol s needs to be represented.

$$I(s) = -\log_2 P(s)$$

Considering these concepts, we developed a program called *cpm*, which can estimate the total number of bits to encode a file, along with the average number of bits per symbol.

1.3 Evaluation methods

To evaluate the performance of our copy model compression, we developed a program called *mutate*, that changes the symbols in a file according to a given probability. Its purpose is to break some patterns in the data, which should affect the compression capability of our copy model approach.

We also compared the performance of our copy model approach with general purpose compression tools, for several types of data and different mutation probabilities.

2 Mutate

The *mutate* program is a simple command-line tool that mutates symbols in each input file according to a probability. The program reads the input file and writes the mutated stream to an output file. The user can specify the seed for the random number generator and the probability of mutation. [3]

2.1 Compile

It is optional since you can use the provided executable inside the **`bin`** directory.

- Run **`cd bin`** in root.
- Run **`g++ -Wall -O3 -o mutate ../src/mutate.cpp`** to compile the program.

2.2 Execute

- Run **`cd bin`** in root to change to the executable's directory.
- Run **`./mutate REQUIRED OPTIONAL`** to execute the program.

2.2.1 Required arguments

- **`-i input_file_path`**: path to the input file (**`string`**).
- **`-o output_file_path`**: path to the output file (**`string`**).

2.2.2 Optional arguments

- **`-h`**: shows how to use the program.
- **`-p probability`**: probability of mutation (**`double, default=0.0`**).
- **`-s seed`**: initialization for the random number generator (**`long or string, default=current_ts`**).

2.2.3 Example

```
./mutate -i ../example/chry.txt -o ../example/chry_mod.txt -p 0.5 -s 1234
```

2.3 Flow of execution

1. Parse program arguments.
2. Read input file and get the alphabet.
3. Set the seed for the random number generator.
4. For each symbol *s* in the input file.

4.1. Get a random double *r* between 0 and 1, consider *p* as the mutation probability.
4.2. If $r < p$.
    4.2.1. Get a random symbol *s'* from the alphabet.
    4.2.2. Append *s'* to the output file stream.
4.3. Else.
    4.3.1. Append *s* to the output file stream.
5. Exit the program.

2.4 Execution on *chry.txt* file

Considering the following configuration of the program, we mutate the *chry.txt* file provided by the professors, with a mutation probability of 0.5 and the seed 1234.

```
./mutate -i ../example/chry.txt -o ../example/chry_mod.txt -p 0.5 -s 1234
```

We can compare the original file with the mutated file to visualize the program's results (first thirty symbols). In red we highlight the symbols that are different from the original sequence.

Original: **GAATTCTAGGCTTTCTTTGAAGAGGTAGTA…**

Mutated: **ACACTCCCGACTTTCGTTGTACAGGGATGA…**

Important note: a symbol can be replaced by itself when it is chosen to be mutated, therefore the number of differences do not directly correspond to the number of mutations.

3 Copy model

As previously mentioned, the aim of this project is not to build a compressor or decompressor, but to estimate the number of bits required to encode a file using a copy model. In this sense, we developed a *cpm* program that estimates the total number of bits to encode a file, along with the average number of bits per symbol. Initially, the program loads the input file to memory by performing a first pass through the text and storing the alphabet and the absolute frequency of each symbol. Then, it performs a second pass through the text with a reading pointer and at least one copy pointer.

3.1 Compile

It is optional since you can use the provided executable inside the **`bin`** directory.

- Run **`cd bin`** in root.
- Run **`cmake .. && make`** to compile the program.

3.2 Execute

- Run **`cd bin`** in root to change to the executable's directory.
- Run **`./cpm REQUIRED OPTIONAL`** to execute the program.

3.2.1 Required arguments

- **`-f input_file_path`**: path to the input file (**`string`**).
- **`-w window_size`**: size of the sequences (kmers) from which the copy model will be enabled (**`int`**).
- **`-n number_of_tries`**: number of predicted symbols to consider when evaluating the miss rate (**`int`**).
- **`-m misses_threshold`**: maximum number of misses allowed in the last **`number_of_tries`** predictions (**`int`**).
- **`-s smoothing_factor`**: parameter to smooth the first probability estimation (**`double`**).
- **`-l limit`**: maximum number of simultaneous copy models (**`int`**).

3.2.2 Optional arguments

- **`-h`**: shows how to use the program.
- **`-o output_file_path`**: path to export the results (**`string`**).

3.2.3 Example

```
./cpm -f ../example/chry.txt -w 15 -s 1.0 -m 7 -n 15 -l 1
```

3.3 Flow of execution

Main

1. Parse program arguments.
2. Read input file and get the alphabet.
3. Initialise a CopyModelRunner object with the passed arguments, the in-memory text, and the alphabet.

CopyModelRunner::constructor

4. Iterate over the text to get the absolute frequency of each symbol and calculate at the same time the necessary bits to encode the greatest counting (*necessaryBits*).
   E.g., if $count('A') = 10$, $count('G') = 6$ and $count('T') = 4$, we only need 4 bits to encode the countings.
5. Calculate the overhead of pre-appending the alphabet and the countings to the compressed text: 8 bits for the alphabet size, 8 bits per symbol, *necessaryBits* bits per counting; add the overhead to the estimated total number of bits.
6. Iterate over the text with a reading pointer (*ptr*), a hash map to store the anchors of each found sequence (*sequenceMap*), a vector to store the active copy models (*currentReferences*), and a vector to store the last predictions of each active copy model (*pastOccurrences*).

CopyModelRunner::runStep

7. Retrieve the current sequence (*sequence*) from the text. If the reading pointer has a lower position than the window size (*windowSize*), the missing elements of the fixed-size window are filled with the first symbol of the alphabet *S*, by imagining $windowSize - 1$ occurrences of *S* before the text.
8. If there are no active copy models yet.
   8.1. If there are no anchors yet ($ptr = 0$), create a new "imaginary" anchor (-1) that points to the first symbol of the alphabet, add it to the *sequenceMap*, and activate the corresponding copy model by pushing it to the *currentReferences* and creating a history entry in *pastOccurrences*.
   8.2. If there are anchors and the current sequence is in the *sequenceMap*, activate the corresponding copy models by pushing them to the *currentReferences* and creating a history entry in *pastOccurrences* for each of them.
   We emphasise that the number of simultaneous copy models is limited by the user-defined limit parameter.
   8.3. If there are anchors, but the current sequence is not in the *sequenceMap*, retrieve the actual char at position ptr (*actualChar*) and use a fallback model by adding to the result

$-\log_2 \frac{counts(actualChar)}{totalChars}$ bits, being *totalChars* the total number of symbols not yet processed.

9. For each active copy model:
   9.1. Predict the next symbol.
   9.2. If hit, add *I(s)* bits to a temporary result stored in the CopyModel object (*partialNumberOfBits*).
   9.3. If miss, add $-\log_2\left(1 - P(hit) \times \frac{counts(actualChar)}{totalChars}\right)$ bits to *partialNumberOfBits*.
   Note that we use the real frequency of the actual char instead of performing a uniform distribution.
   9.4. Update the history of the copy model with the result (hit/miss).
   9.5. If in the last *numberOfTries* predictions, there are more than *missesThreshold* misses, deactivate the copy model.
      9.5.1. Remove the copy model from the *currentReferences* and delete its history from *pastOccurrences*.
      9.5.2. If it is the last standing model of the sequence (last survivor), add its *partialNumberOfBits* value to the final result.
10. Decrement the count of the already processed *actualChar* and *totalChars*.
11. Add a new anchor with the current *ptr* to the sequence entry in hash map.
12. Stop when the reading pointer reaches the end of the text.

CopyModelRunner::addRemainingBits

13. Add to the final result the *partialNumberOfBits* of copy models that survived until the end of the text.

Main

14. Display the estimated total number of bits and the average number of bits per symbol.
15. Export the results to a file (optional).

3.4 Decisions made

Decision	Justification
A *runStep* method to execute only one iteration of the algorithm, instead of having the whole algorithm main cycle in one CopyModelRunner method.	Support for parallel execution of multiple instances of CopyModelRunners with different window sizes, thresholds, and limits of simultaneous copy models, such that their partial results could be compared and eventually combined.
Unordered map instead of map to store the anchors.	While map has a complexity of O(log n) for insertion, unordered_map has a complexity of O(1) for insertion.
Miss rate in recent predictions instead of a hit percentage threshold.	Deactivating a copy model when it exceeds *missesThreshold* misses in the last *numberOfTries* predictions proved to be more effective than deactivating it when it falls below a hit percentage threshold.

Criteria to choose a copy model when multiple models are active: the last surviving model.	Several criteria could be used to choose a copy model (the most accurate, the most recent, the oldest, etc.). Through empirical tests, we verified that the last standing model is the one that fails the least, and therefore, it contributes to a lower number of bits in the final result.
Decrement the count of the already processed chars.	Like using a uniform distribution, calculating the probabilities with the original absolute frequencies of the symbols is inaccurate and does not reflect the real distribution of the text at a given moment of the algorithm.

4 Parameter using the chry.txt file

Research in optimisation problems have shown that carrying out a random search for the best parameters can be more efficient than a grid search, i.e., testing all combinations of parameters, when the configuration space is very large. [4] Given the high number of parameters to test, in addition to the already mentioned methods, we performed a random search for each parameter.

Leveraging the provided *chry.txt* file, we proceed to make several tests where we variated 1 variable and fixed the others to find the best parameters (parameter tuning). The default parameters were:

- Window size: 15
- Lookback size: 15
- Misses threshold: 7
- Smoothing factor (alpha): 1
- Limit: 1

4.1 Window size

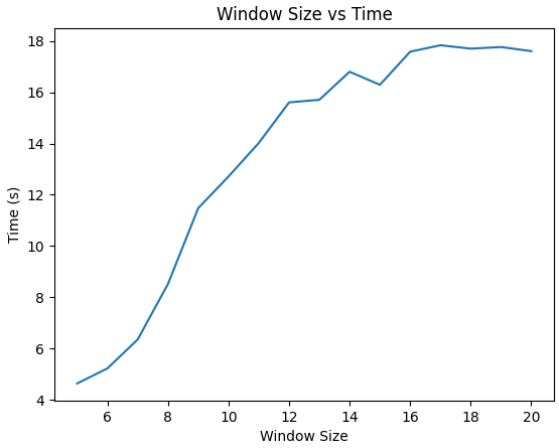

Figure 1. Time (s) as a function of the window size.

In this figure, we see that the time the copy model takes increases when the window size also increases. This can be explained because there is more data to be managed each time.

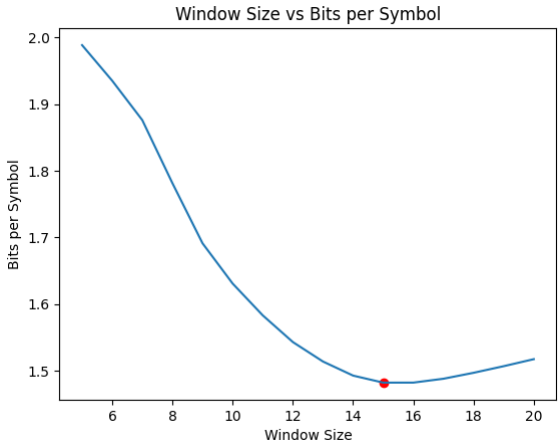

Figure 2. Bits per symbol (bps) as a function of the window size.

In this figure, we see that the bits per symbol firstly decreases until the window size is 15 and after that starts to increase. This occurs because firstly because a larger window allows the compression algorithm to find longer repeating patterns in the data, longer patterns can be represented more efficiently with fewer bits compared to shorter patterns. However, after a point starts to increase because the overhead of maintaining and referencing the window also increases and additionally the likelihood of finding longer repeating patterns may decrease as the data further from the current position becomes less relevant.

4.2 Lookback size

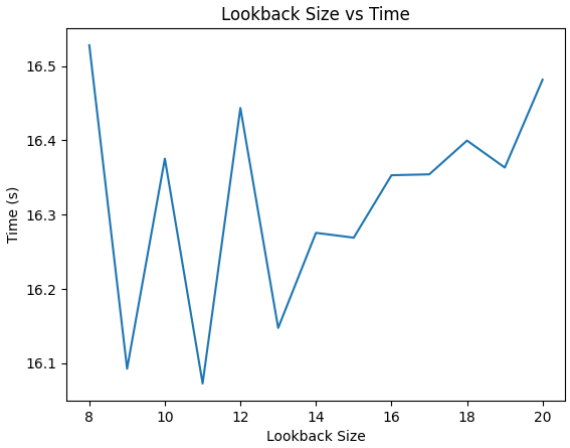

Figure 3. Time (s) as a function of the lookback size.

In this figure, we see that the time varies too much, therefore we can conclude that the lookback size does not appear to have influence in time that takes for the copy model to run.

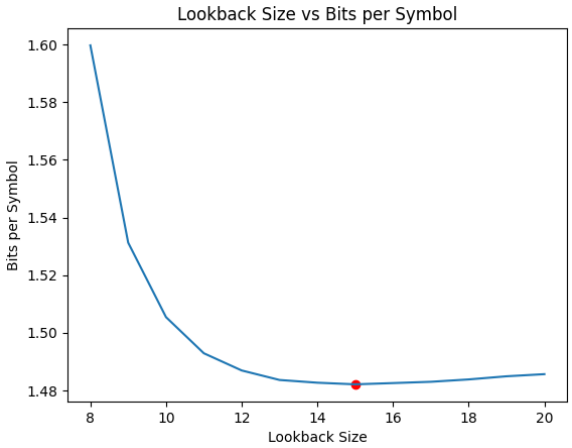

Figure 4. Bits per symbol (bps) as a function of the lookback size.

In this figure, we see that the bits per symbol decreases when the lookback increases but from a certain point (when lookback size is 15) it starts to increase. This is because with a too big of a lookback, the copy model has a bigger probability of failing multiple times (more than ideal).

4.3 Misses threshold

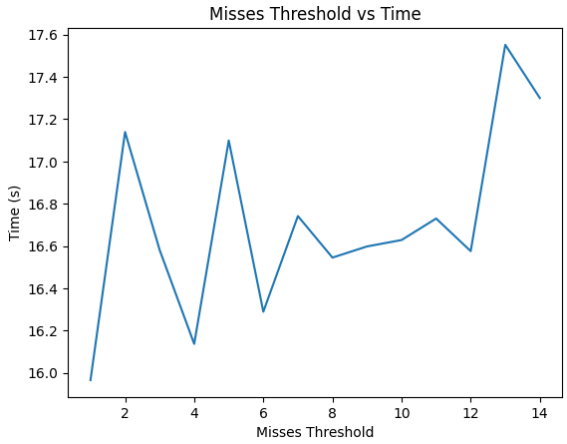

Figure 5. Time (s) as a function of the misses threshold.

In this figure, we can see that the time varies too much, therefore we can conclude that the misses threshold does not appear to have influence in the time that takes for the copy model to run.

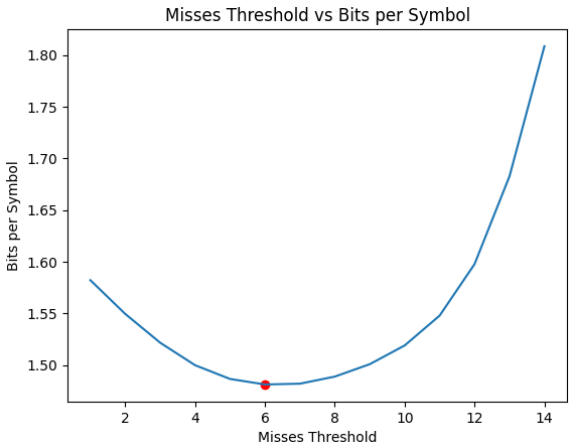

Figure 6. Bits per symbol (bps) as a function of the misses threshold.

In this figure, we can see that the bits per symbol first decreases with the increase of the misses threshold, but from a point it starts to increase a lot. This happens because until it starts to increase (when misses threshold is 6) we are adopting a rigid politic of control which only lets the copy model miss a few times. After that, we are letting the copy model miss too many times.

4.4 Smoothing factor (alpha)

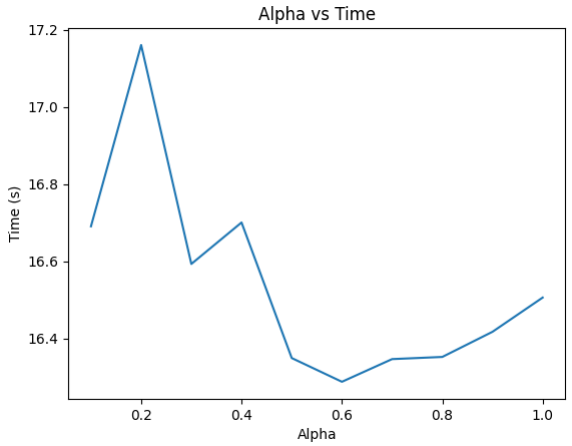

Figure 7. Time (s) as a function of the smoothing factor (alpha).

In this figure, we can see that the time varies too much, therefore we can conclude that the smoothing factor (alpha) does not appear to have influence in the time that takes for the copy model to run.

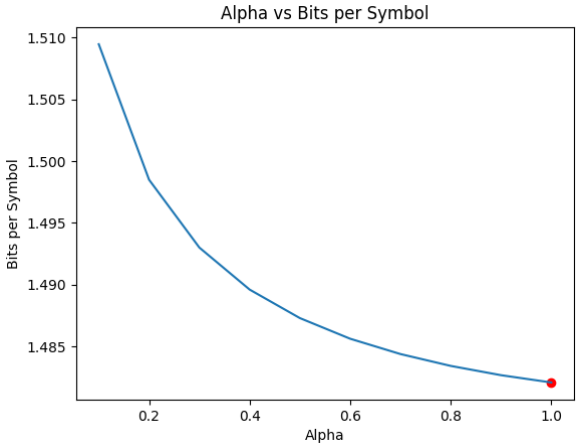

Figure 8. Bits per symbol (bps) as a function of the smoothing factor (alpha).

In this figure, we can see that an increase of the smoothing factor (alpha) decreases when the bits per symbol decreases. This is because the increase of the smoothing factor often enables the algorithm to better generalize, remove redundancy, make more accurate predictions, and optimize symbol representation.

4.5 Limit

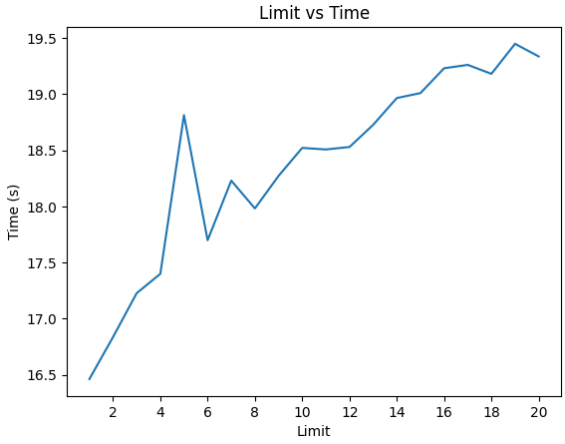

Figure 9. Time (s) as a function of the limit.

In this figure, we can see that as the limit increases, the time that takes to the copy model to run also increase (with some exceptions).

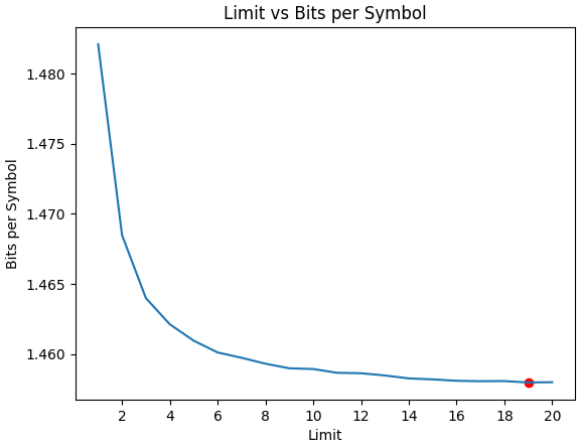

Figure 10. Bits per symbol (bps) as a function of the limit.

In this figure, we can see that as the limit increases, the bits per symbol decreases until it stabilizes. This illustrates the benefit of using several copy models simultaneously.

4.6 Conclusion

Using the random search method, we conclude that the best parameters are:

- Window size: 16
- Lookback size: 15
- Misses threshold: 6
- Smoothing factor (alpha): 1
- Limit: 19

5 Comparative study

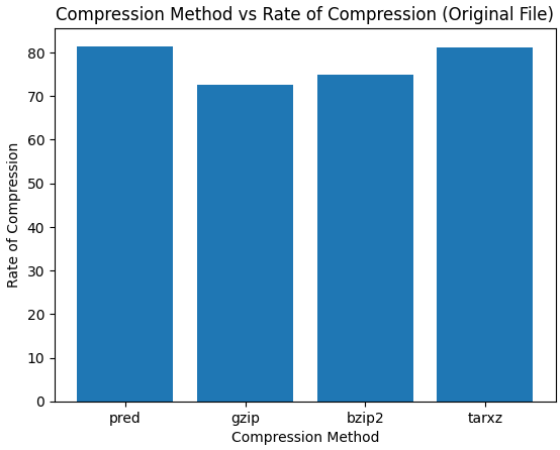

Figure 11. Rate of compression for each compression method.

In this figure, we can see that our copy model does not perform as well as some of the most used compressors, however it comes close to the *tarxz* compressor.

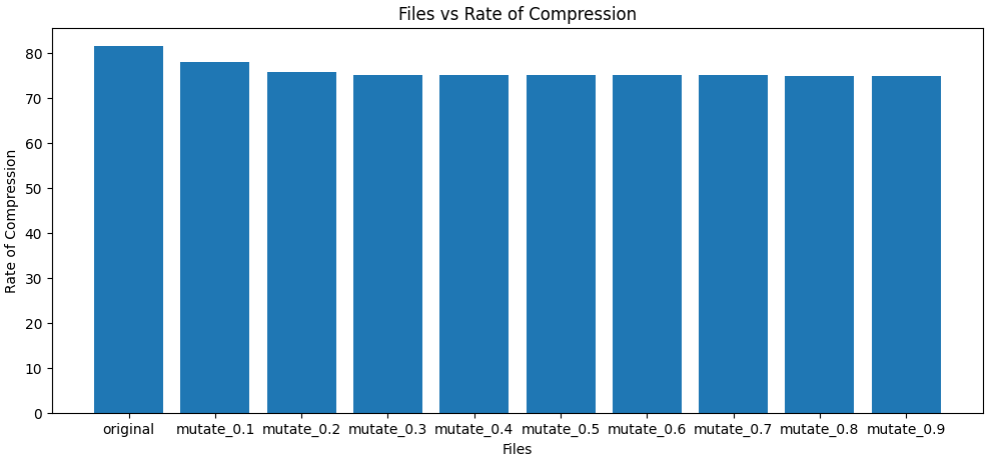

Figure 12. Rate of compression for each mutated version of chry.txt.

In this figure, we can conclude that the higher the probability of mutation, the higher is the rate of compression is using our copy model.

6 Conclusion

This assignment allowed us to explore how copy models can be applied to genomic data compression. We implemented a program that used copy models and was able to estimate the total number of bits to encode a file, along with the average number of bits per symbol. To evaluate its performance, we developed a program to randomly mutate symbols in a file, with the goal of breaking existing patterns. Finally, we compared our copy model compression approach to existing general-purpose tools.

References

[1] A. J. Pinho and D. Pratas, "eLearning@UA," February 2024. [Online]. Available: https://elearning.ua.pt/course/view.php?id=5431. [Accessed February 2024].

[2] A. J. Pinho, D. Pratas e S. P. Garcia, "GReEn: a tool for efficient compression of genome," *Nucleic Acids Research,* vol. 40, nº 4, p. e27, 2012.

[3] D. Paiva, J. Fonseca and R. Gonçalves, "TAI Assignment 1," GitHub, February 2024. [Online]. Available: https://github.com/joaompfonseca/tai-assignment1-group8. [Accessed March 2024].

[4] J. Bergstra e Y. Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research,* vol. 13, pp. 281-305, 2012.