Copy and Generator Models for Data Compression

Teoria Algorítmica da Informação
Ano letivo 2022/2023

Dinis Santos Lei, nMec 98452
João Pedro Saraiva Borges, nMec 98155
Vicente Samuel Gonçalves Costa, nMec 98515

Universidade De Aveiro

Index

Introduction

Data compression is the process of reducing the size of a data file by encoding its information in a more efficient manner. The use of compression algorithms has become increasingly important with the explosion of data storage and transmission needs in various fields such as multimedia, image processing, and internet applications. The Copy Model is a data compression technique that has gained popularity due to its simplicity and effectiveness. This technique has been used in various applications such as text compression, image compression, and DNA sequence compression.

Copy Model

The copy model is a data compression technique that tries to predict the next outcome by looking at a reference in the past.

The model estimates the probability of the next symbol being the same as its past reference, and based on that prediction it can calculate the amount of information gained.

The probabilities are calculated by having counters of hits (Nh) and misses (Nm). These counters will give the relative frequency Nh/(Nh+Nf). However, due to the problem of giving 0 probability to events that haven't appeared in the model, for example, the first time an event is calculated, a smoothing parameter alpha is introduced.

With the probability calculated we can then estimate the amount of information generated by the new symbol, s, -log2(P(s)).

First consider a sequence of output symbols generated from the alphabet E = {A, C, G, T}.

To start, the copy model will grab the last k symbols (window).
Then it will fetch a pointer to one position where the window has appeared in the past, we call this an anchor.
Now the anchor, based on the number of hits and misses (initially 0), will give a probability that the next symbol in the output will be the same as the next in its sequence.
If the prediction is correct the amount of information gain is what was stated in the formula above.
But if the prediction is incorrect, the amount of information gained is [formula].
The last steps are repeated until the performance of the anchor, that is, the prediction probability becomes lower than a given threshold.
When that happens a new window will be selected and a new anchor will be chosen to start the copy model.
In figure X. we can see the copy model acting with a window size k = 4 and alpha smoothing = 1.

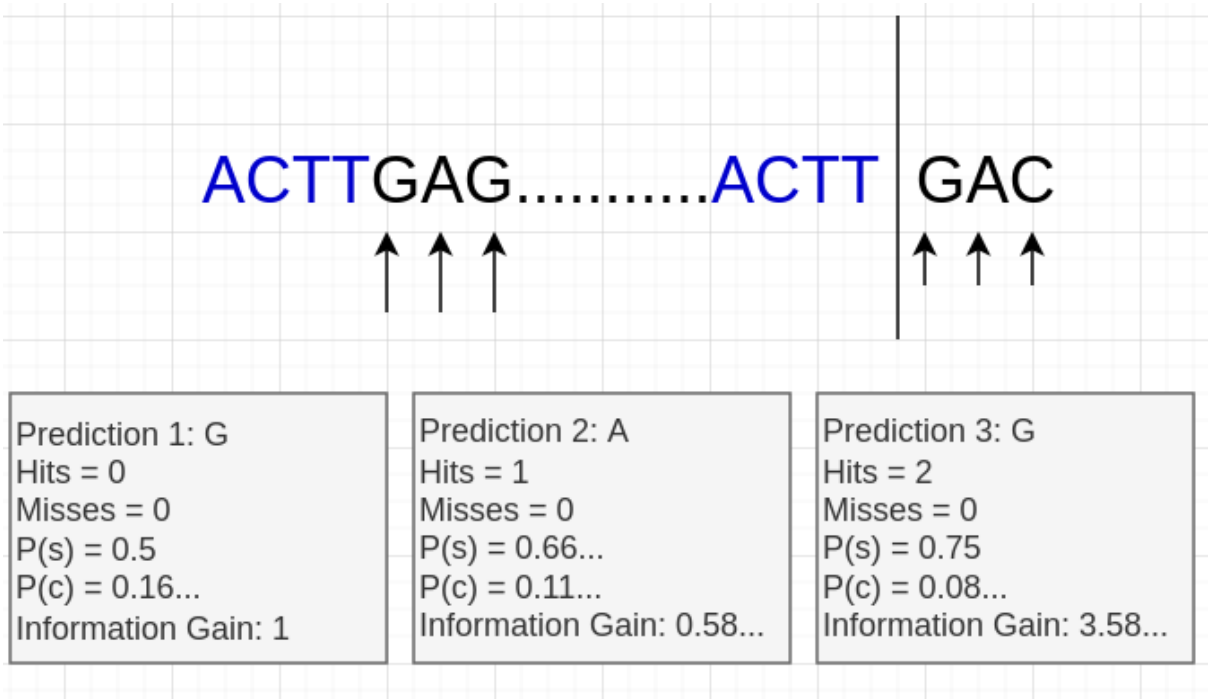

Figure 1. Example of Copy Model in action

Changes to the Copy Model

Our version of the copy model allows to run the standar copy model while changing the parameters of window size (k), smoothing agent alpha (a) and the probability threshold (t), but it also allows to run with 3 other parameters, minimum size (m), number of anchors (a) and ignore last guess (i).

Minimum Size

This parameter tells the copy model to ignore the first m symbols before evaluating its accuracy. The idea is to try to minimize the impact of the first guesses. As we can see in the example file *“examples/min_size_example.txt”*, running the model with m = 1 will have a better performance than running with m = 0.

Number of Anchors

This parameter is the number of active anchors supported on a given window. Instead of choosing the last occurrence of the window in the sequence to start the copy model, we select the n-last occurrences and run the copy model simultaneously. Then after every anchor has crossed the shutdown threshold, we select the one who encoded the most symbols.
This approach should significantly increase the running time of the copy model but in turn will increase its performance

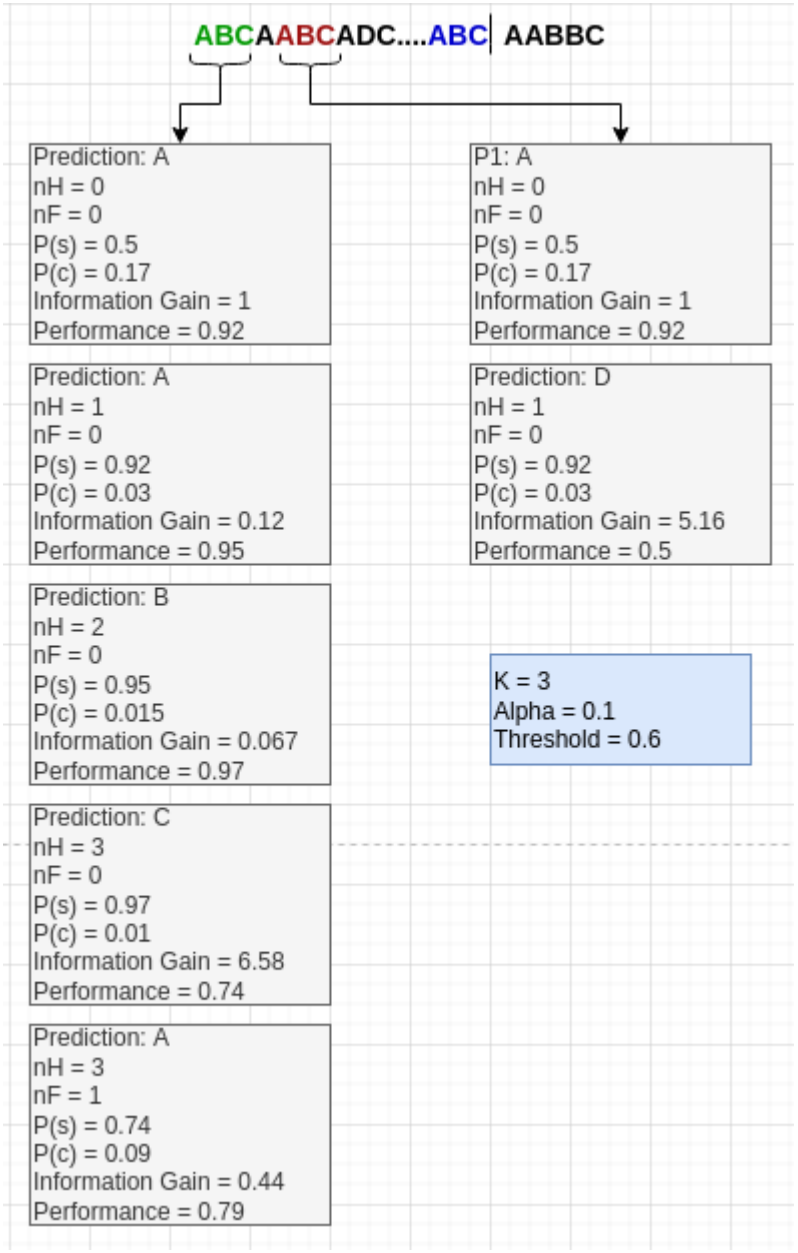

Figure 2. Multiple Anchors example.

Ignore last guess

This parameter makes the copy model shutdown one symbol earlier. This is based on the fact that the last symbol the copy model encodes usually represents a miss, and encoding a miss generates more information than encoding without the copy model.

Results

To evaluate the copy model and find out the set of the best hyperparameters, we ran the copy model on the DNA chromosome example *“example/chry.txt”* while changing the following parameters:

- k (window size) = {8, 10, 12, 14}
- m (minimum size) = {0, 1}
- a (alpha) = {1, 0.1, 0.01}
- t (threshold) = {0.5, 0.7, 0.8}
- n (number of anchors) = {1, 4, 8}
- i (ignore last) = {On, Off}

In total 432 data points were collected. The data was divided in runs with or without the ignored last guess flag because the effects of this flag are more unpredictable. The data sets were evaluated on the following 2 metrics, the **average information** given by a new symbol and the **run time** of the copy model.
The results are present on the jupyter notebook visualization.ipynb.
From the data we can confirm the hypothesis proposed in the section *Number of Anchors*, the number of anchors is directly proportional to the execution time, and in fact seems like the factor that contributes the most, and also has a major impact on decreasing the average information.
The minimum size can have an impact on the average information, but the advantages seem to vary with the other hyperparameters.
The ignored last guess produces an overall lower average information.
Lastly for this specific file the best values for k, alpha and threshold are, 8, 1 and 0.8 respectively.
In figure 3 we can see the instant information, of the first 5000 steps, of the best runs with and without the ignored last guess flag.

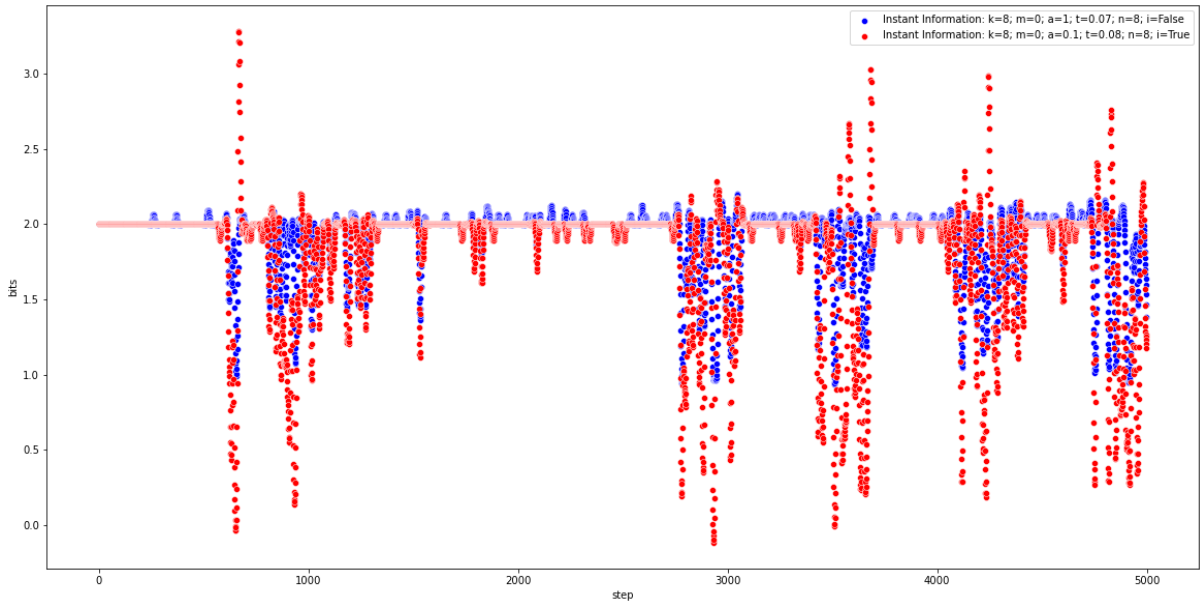

Figure 3. Instant Information for the first 5000 steps of the best hyperparameters.

Generator

For the generator model, our objective is to develop a model which is able to take a file with symbols for training purposes, take an initial sequence of symbols as an input, and generate a sequence of symbols following the model file given as training.

First Approach

As a starting point, we would read the whole training file and create a map with the sequence of symbols as a key, and another map as value, with the key of it being the next symbol shown in the sequence and the value the amount of times this symbols was shown, there is also an entry in the map which has as key “total”, which has the amount of times the sequence appeared in total.

Subsequently, the program calculates the probabilities for all the symbols that might appear in each sequence, then, if a sample input is given to the program, the program will use it as the initial sequence of symbols and check if the sequence is in the map, if not, the program does search, where it takes a random generated number between 0 and 1, and with the cumulative probability of all the symbols in the dictionary, the program sees where the random number falls in the cumulative probability of all symbols; take a casino roulette as example, where each of the slots in the roulette represents a symbol, and they vary of size according to their probability; instead of being a roulette, the probabilities are summed up until the random generated number is the result, we take the lower bound of the sum and the last symbol added to the sum is the symbol that we have obtained.

This method is also used when a sequence is found in the map, instead of taking the whole dictionary of symbols, we take the possible next characters that appeared in the training file with the given sequence, and calculate the next symbol as explained before. We should note that in both approaches, for better generated results, you should take an excerpt of the training file and give it as a sample to the generator, so that it makes concise results according to the context given (training file).

Results

To evaluate the first approach to the copy model generator, we ran this program with the following parameters:

- k (window size) = {3,4,5,6,7,8,9,10,11}
- n (number of generated symbols) = 100
- train file = *“bible.txt”* (4MB)
- sample text = {“LOR”, “name”,”stone”, *“Heaven”,”heavens”,”fruitful”,”abundantl” ,”compasseth”,”aaaaaaaaaaaa” }*

For k=3, the program took 0,726 seconds to run and generated the following result:

- “(LOR)D, ance artion, I upon he secraffering of mand morn of Ephraighteremit upon the women of Israel, I h”

For k=4, the program took 0,958 seconds to run and generated the following result:

- “(name) top of find he which is is would his own thee unto him, Thou hast of the Chaldeansed, and two offer”

For k=5, the program took 1,684 seconds to run and generated the following result:

- “(stone), and saw heart though all ye believed not be fulness and of Israel hath carried against he made nig”

For k=6, the program took 1,973 seconds to run and generated the following result:

- “(Heaven) and was not been done intelligence;

Behold, thy rod, are for my name, until the sight, and Gaius;

For k=7, the program took 2,449 seconds to run and generated the following result:

- “(heavens), and turn aside them as a dumb man thing, it was wroth, and said unto him: so it was to burn it up,”

For k=8, the program took 3,017 seconds to run and generated the following result:

- “(*fruitful*), and we shall they knew not the people, and not as an harlot; because ye know how to speak unto the”

For k=9, the program took 3,566 seconds to run and generated the following result:

- “(*abundantl*)y: thou shalt beget sons and daughter of Jerusalem stood round about: the length thereof trembling a”

For k=10, the program took 3,890 seconds to run and generated the following result:

- “(*compasseth*) me about their armies together the pillars three, and they will keep it unto life.

Verily, verily,”

For k=11, the program took 4,100 seconds to run and generated the following result:

- “(aaaaaaaaaaa) m tg trnA, fhtefhee aec h,l ssoe e ys a ennadhnt e fstdysfe ayrRdayfnmeo c Onyaatoetwnedd o et: a”

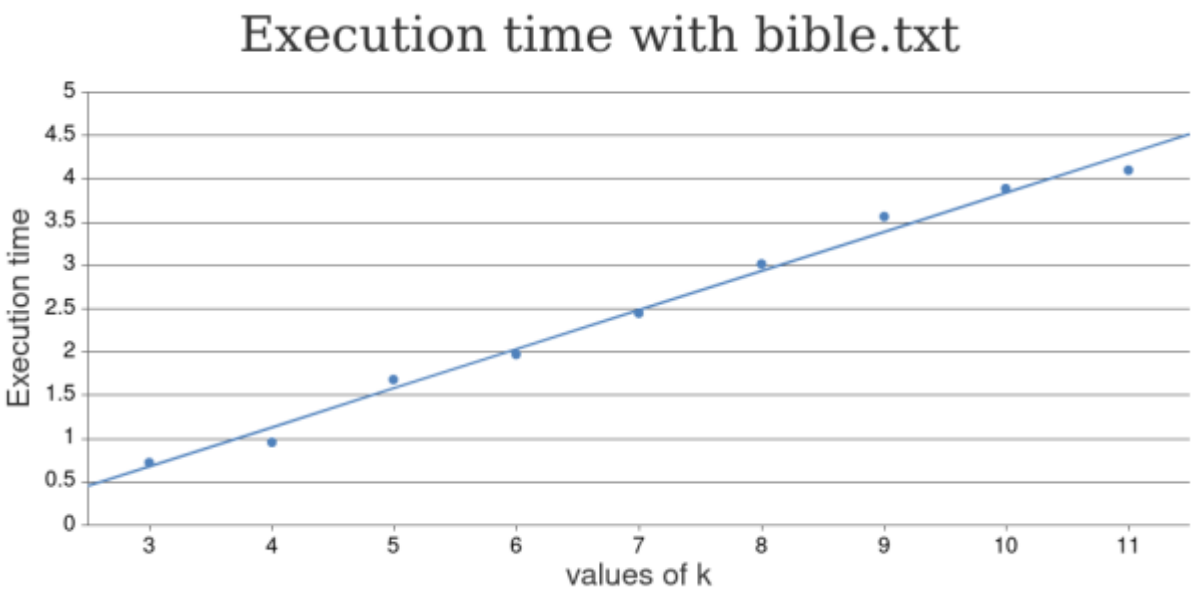

Figure 4. Graph with execution time with “bible.txt”

The results given by the generator are satisfactory, it is fast to process the training file and also fast to generate a new sequence of symbols (as visible in figure 4), however when given a sample text that was not on the original file, the program will generate an incoherent sequence (example with k=11 and sample text “aaaaaaaaaaa”).

Next, we ran the program with a bigger file, the *“chry.txt”* (22mb), with a value of k varying from 1 to 29 and saved the execution time in the figure 5.

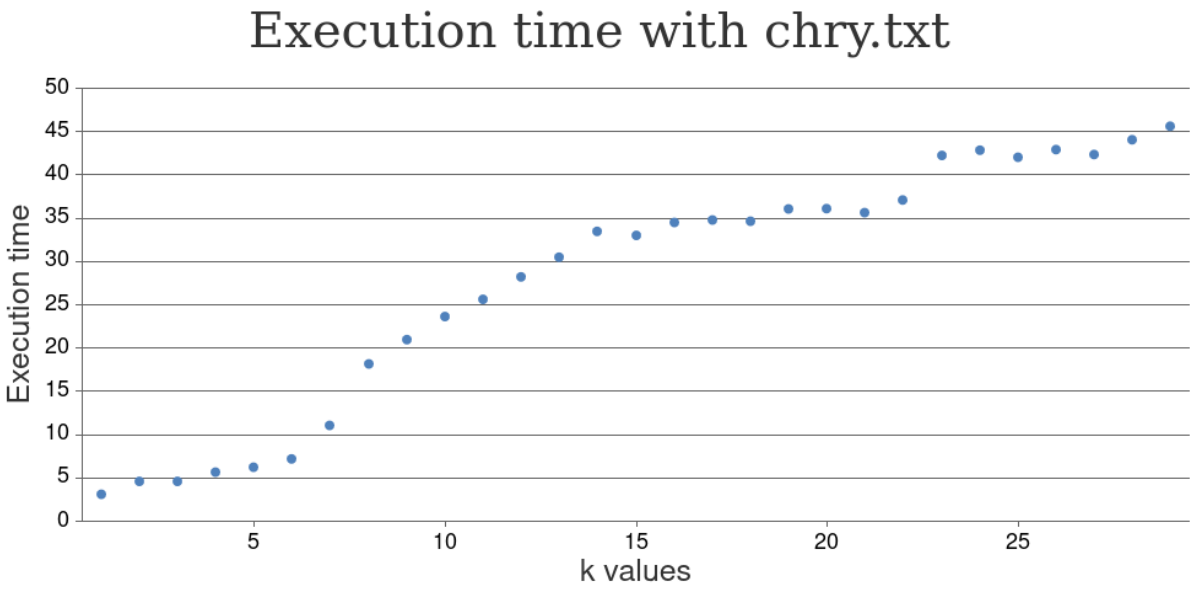

Figure 5. Graph with execution time with “chry.txt”

As seen in this figure, the execution time has a linear increasing behavior with the increasing of the “*k”* values. Also, we can also conclude that the size of the file has a negative impact on the execution time since the execution time of the program with the chry.txt file (22MB) is bigger than the bible.txt (4MB).

Second Approach

We also made a second version of the generator, entitled cmp_gen2, where we took a different implementation. We maintained the approach of choosing the next symbol according to a random generated number, and, instead of calculating all the probabilities of the next symbols according to a given sequence right after reading the whole training file, we only save the positions where a certain sequence ends, and save it in a map that takes the sequence as key, and saves as value a list with all the indexes where the corresponding sequence ended. We only calculate the probability of each of these symbols occurring in a certain sequence if this sequence does appear in the generated sequence being made. This approach also obtained as good results as the previous models, and probably even doing better with bigger training files.

Results

We run this program with the following parameters:

- k (window size) = {4,5,6}
- n (number of generated symbols) = 100
- train file = *“bible.txt”* (4MB)
- sample text = {“name”,”stone”, *“Heaven” }*

For k=4, given the sample “name”, the program generated the following result:

- “(name) to the Levites are he daught throught in learn as man.

If the reach then wenty them, When I stand”

For k=5, given the sample “stone”, the program generated the following result:

- “(stone).
  Then Jacob: And thee; (for him that is, Bethshemesh.
  And Solomon, and in his people inhabitation”

For k=6, given the sample “Heaven”, the program generated the following result:

- “(Heaven) over the LORD giveth on a certainly returned in Jerusalem, and speak, and sleep in his ring of Egyp”

Next, we ran the program with *“chry.txt” file* and a value of k varying from 1 to 20. We saved the execution time in the figure 6.

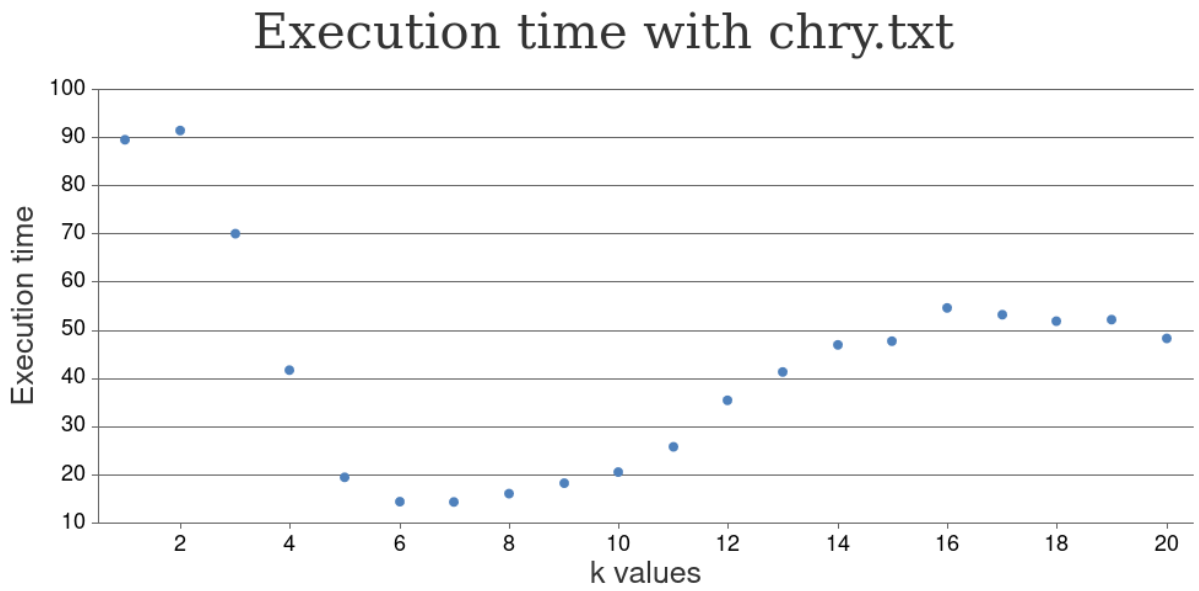

Figure 6. Graph of cpm_gen2 execution time with “chry.txt”.

As seen in the results above, the second approach also produces satisfying results giving frequently coherent words. However it has a complex execution time graph. While the first approach has a linear increase in execution time, this approach, as seen in figure 6,has an execution time that is drastically high with “*k”* value of 1 and decreases until “*k”* equals 6, so it can begin to rise until “*k”* equals 16, then slowly decrease.

Results conclusion

In conclusion, both approaches are valid solutions with good results, however when working with smaller “*k”* values the first approach is optimal due to the execution being lower while with bigger *“k”* values the second approach should have a better performance.

Conclusion

Overall, we obtained not only good response results in both the copy model and generator, but also good performance results; we believe that some optimizations are still missing in the generator program, such as when not given a sample, the program should not pick a random sequence of symbols according to the probability of each symbol, but a sequence caught in the training file; and we do believe that this influences a lot the final performance result, specially in the second generator. However, we believe that we fulfilled the objectives of this project with decent results.