Source coding for *the Game of Thrones*

Second-order Markov Adaptive Approximation, Huffman and Fano Coding

Name: Jinming Ren, Yuhao Liu
UESTC ID: 2022190908020, 2022190908022
UofG ID: 2840216R, 2840218L
Date: November 28, 2024

University: UoG-UESTC Joint School
Location: No.6 Research Build A240

Contents

List of Figures

List of Tables

Abstract

In this article, we successfully implemented huffman and Fano coding to encode/denode the first three chapters of *the Game of Thrones* and evaluated their raw performance in terms of the average code length, code rate, efficiency, and compression ratio in MATLAB. We also presented a new lossless coding scheme called 2nd-order Adaptive Markov Encoding (2nd-ord AME, abbreviated AME) coding and evaluated the overall performance when combined with huffman coding and fano coding.

1 Introduction

Source coding, a fundamental technique in data compression, plays a vital role in modern information transmission and storage. Its primary objective is to reduce the number of bits required to represent symbols from a source by leveraging their inherent statistical properties. This efficiency enables effective data storage and transmission while maintaining the integrity of the original information.

In this project, we explore the application of Huffman Coding and Fano Coding, two widely recognized entropy-based compression techniques. Huffman Coding, a lossless algorithm, is designed to assign shorter codes to frequently occurring symbols, making it highly efficient for sources with uneven symbol distributions. On the other hand, Fano Coding, while similar in principle, uses a different approach to partition symbols based on frequency.

Using MATLAB, we implemented both methods to encode and decode the first three chapters of a selected text, generating metrics such as average code length, code rate, efficiency, and compression ratio for comparison. Additionally, we introduced a important change-of-perspective that compression is essentially the same as prediction. Based on this idea, we introduced a novel pre-processing technique, Second-order Adaptive Markov Encoding (AME), designed to enhance the compression performance by uncovering structural patterns in the source text. The efficiency of AME was also evaluated.

2 Compression is Equivalent to Prediction

In this section, we state the motivation behind AME coding.

Both huffman coding and fano coding are based on the assumption that the source is *discrete memoryless*. They are trying to approach the compression limit (which is the entropy of the source [1]) given that the original source looks random. So they are called "entropy coding". However, in practice, the texts inherently take some structures that cannot be explicitly captured by any relative simple models. These structures are not considered when performing entropy coding, so we wasted some potential compression ratio.

No matter what coding method we choose, such as LZ77 [6, 7], the last step them is always entropy coding. But entropy coding is well-understood and mature (Huffman being the "optimal" in some sense). Therefore, the only way to improve compression ratio is to uncover the structure of the source. We need to design a better model first to capture and hiding those structures while exposing the true "memoryless components" of the source, then the utility of entropy coding can be maximized.

The basic idea behind AME coding is that compression is equivalent to *prediction*. We are essentially building a same text predictor in both sides of the transmitter and the receiver. As long as the predicted next character matches the true one, that character is not considered as the "memoryless component" of the source (it depends on the history text). But if the predicted one doesn't match the true one, this means there are somewhat "random" factors come into play. If we can proposed some encoding strategy that makes the predictor works exactly the same

on both the transmitter and receiver, we only need to transmit the "memoryless components" and the number of correct predictions.

For example, if we want to extract the "memoryless components" of the following content:

```
Information theory is interesting.
```

Feed it into the predictor, suppose we can correctly predict the character in position 2, 5-11, 15-19, 21-22, 26-34, we only need to transmit the initial letter that cannot be predicted together with the number of correct predictions:

```
I1fo7 th5i2int9
```

The rest letters are somehow losed some internal dependence and can be approximately considered as memoryless.

We can see in this example the efficiency of compression directly relies on how well we extract the "memoryless components" of the text, in other words, the performance of the predictor. Since human language is the product of human mind, which definitely cannot be modelled using just a few parameters. An accurate predictor would need millions of parameters, which is essentially a neural network [5]. However, due to time and space complexity, using a neural network to predict the next character is not feasible in practice. However, we can use a simpler model called markov chain to capture some of the structures in the text.

3 Mechanism of AME Coding

We assume no prior knowledge of the source. So the predictor is built simultaneously with the encode/decode process. The algorithm for AME encoding and decoding are shown in Algorithm 1 and 2 in Appendix A. The codes are written in Python and can be found in the appendix B and C.

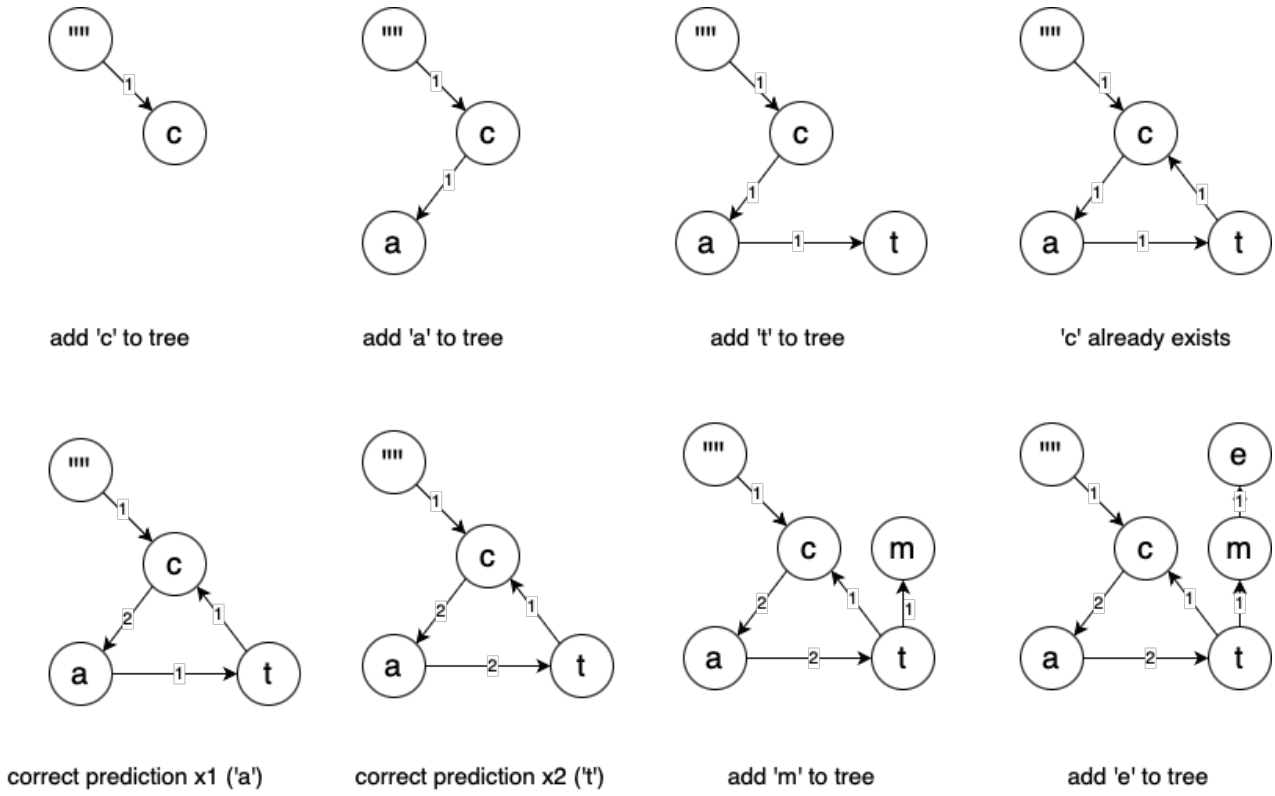

Figure 1: The AME tree construction precedure

We will take an example to show how AME works. Let's consider the following text:

```
catcatme
```

As shown in Figure 1.The first character is `'c'`, so we add `'c'` to the AME tree, mark the weight 1 and add it to the output buffer. The next character is `'a'`, so we add `'a'` to the AME tree, mark the weight also 1 and also add it to the output buffer. Repeat this process until we could possibly "predict" the next character. The first correct prediction is the 5th character `'a'`. The next prediction (`'t'`) is also right. But the character after that is `'m'` and `'e'`, which are not in the tree, thus we then add them to the tree. Therefore, the output buffer is:

```
catc2me
```

It saves one character in this case, but it can be more significant in a larger text. Part of the AME-encoded `original.txt` in shown in Appendix I.

4 Entropy Coding Review

4.1 Mechanism of Huffman coding

Huffman coding is a lossless data compression algorithm proposed by David A. Huffman in 1952 [1]. It reduces the size of data by assigning variable-length codes to characters based on their frequencies of occurrence [2, 3]. The core principle behind Huffman coding is to assign shorter codes to more frequent characters and longer codes to less frequent ones. The process begins with the construction of a frequency table, which lists each character and its corresponding frequency in the data. From there, a binary tree is built by repeatedly combining the two nodes with the lowest frequencies into a new node. This is the clever part, it builds from the ground up (instead of from the node to the leaves). This process continues until all nodes are merged into a single tree, with the root node representing the entire data set. The structure of this tree enables the generation of optimal, prefix-free binary codes, where each character's code is determined by its position in the tree. In the final step, the original characters in the data are replaced with their corresponding Huffman codes, achieving the desired compression. We perform this process in MATLAB and we visualize the tree in the figure in Appendix D.

4.2 Mechanism of Fano coding

Fano coding is another lossless data compression algorithm that assigns variable-length codes to characters based on their frequencies, similar to Huffman coding. The key principle behind Fano coding is to recursively divide the data set into two parts, ensuring that the frequencies of the characters in each part are as balanced as possible, and then assign binary codes accordingly. The process begins with constructing a frequency table, listing each character alongside its frequency. The next step is to split the set of characters into two subsets, aiming to balance the total frequencies of both subsets. Each subset is then assigned a binary digit (0 or 1), and the process is repeated for each subset. As the binary tree is built, characters are assigned codes based on the path from the root to the leaf node representing that character. Finally, the original characters in the data are replaced by their corresponding Fano codes, resulting in compression. Like Huffman coding, Fano coding ensures that the encoded data is more compact without any loss of information. However, its encoding efficiency is quite low when dealing with source symbols that have relatively uniformly distributed probabilities.

All MATLAB codes of the encode/decode process of Huffman and Fano coding can be found in the appendix E and F.

5 Implementation

We performed two seperated experiments for different purposes. The first is to realize Huffman and Fano coding on the original text and perform evaluations. Given that Huffman coding is the optimal method for generating codes, especially when there are significant differences in the probabilities of occurrence for different source symbols, it should significantly outperform the Fano coding technique in terms of coding effectiveness [3, 4]. The second is to combine AME coding with Huffman and Fano coding and evaluate the performance of the new scheme.

5.1 Without AME

The encode/decode process in shown in the form of a flowchart shown in Figure 2.

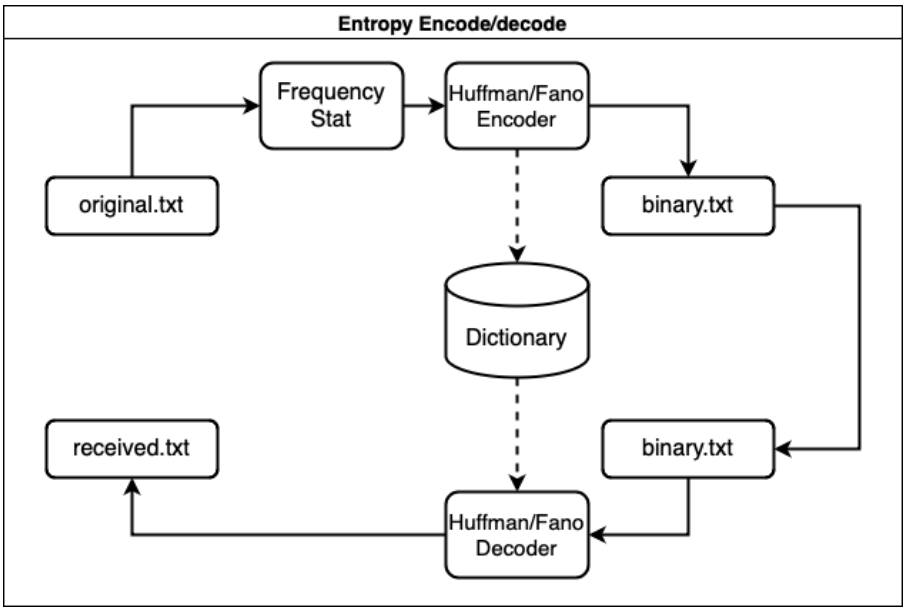

Figure 2: Flowchart of the experiment without AME

The content in `original.txt` contains the first three chapters of *the Game of Thrones*. Part of the `original.txt` is shown in Appendix G.

In order to perform huffman/fano coding on the text, we first need to calculate the frequency of each character in the text. The frequency of each character is shown in Table 1.

Table 1: Frequency of each character in `original.txt`

Index	Symbol	Probability	Index	Symbol	Probability	Index	Symbol	Probability
1	(LF)	0.0052	2	(CR)	0.0052	3	(Space)	0.1915
4	!	0.00020423	5	,	0.0127	6	-	0.001
7	.	0.0165	8	1	2.0423e-05	9	2	2.0423e-05
10	:	4.0846e-05	11	;	0.00024507	12	?	0.002
13	A	0.0013	14	B	0.0017	15	C	0.00055141
16	D	0.00036761	17	E	0.00038803	18	F	0.00079649
19	G	0.0011	20	H	0.0024	21	I	0.0026
22	J	0.001	23	K	0.00020423	24	L	0.00055141
25	M	0.00051057	26	N	0.0011	27	O	0.00046972
28	P	0.00022465	29	R	0.0017	30	S	0.0014

Index	Symbol	Probability	Index	Symbol	Probability	Index	Symbol	Probability
31	T	0.0037	32	U	0.00014296	33	V	8.1691e-05
34	W	0.0034	35	Y	0.00061268	36	a	0.0585
37	b	0.0109	38	c	0.0135	39	d	0.0414
40	e	0.0959	41	f	0.0156	42	g	0.016
43	h	0.052	44	i	0.0435	45	j	0.00061268
46	k	0.0079	47	l	0.035	48	m	0.0154
49	n	0.0476	50	o	0.0546	51	p	0.0079
52	q	0.0004493	53	r	0.0451	54	s	0.0475
55	t	0.0606	56	u	0.0174	57	v	0.0051
58	w	0.0192	59	x	0.00028592	60	y	0.0139
61	z	0.00036761	62	’	0.0026	63	“	0.0051
64	”	0.0051	65	.	4.0846e-05			

According to the frequency of each character, we can create the Huffman/fano tree based on the methods in Section 4.1 and 4.2 and generate the huffman/fano dictionaries[1]. We then use this dictionary to encode the text and calculate the four main metrics including average code length $\bar{L}$, code rate R, efficiency η, and compression ratio ξ. The results are shown in Section sec:result.

Then we constructed the encoded text in binary form (`binary.txt`). Part of the content of (`binary_huffman.txt`) is shown in Appendix H. Then we send the encoded text (`binary.txt`) to the receiver and decode it using the same dictionary to recover the original text.

5.2 With AME

The experiment procedure of the encode/decode process in shown in Figure 3.

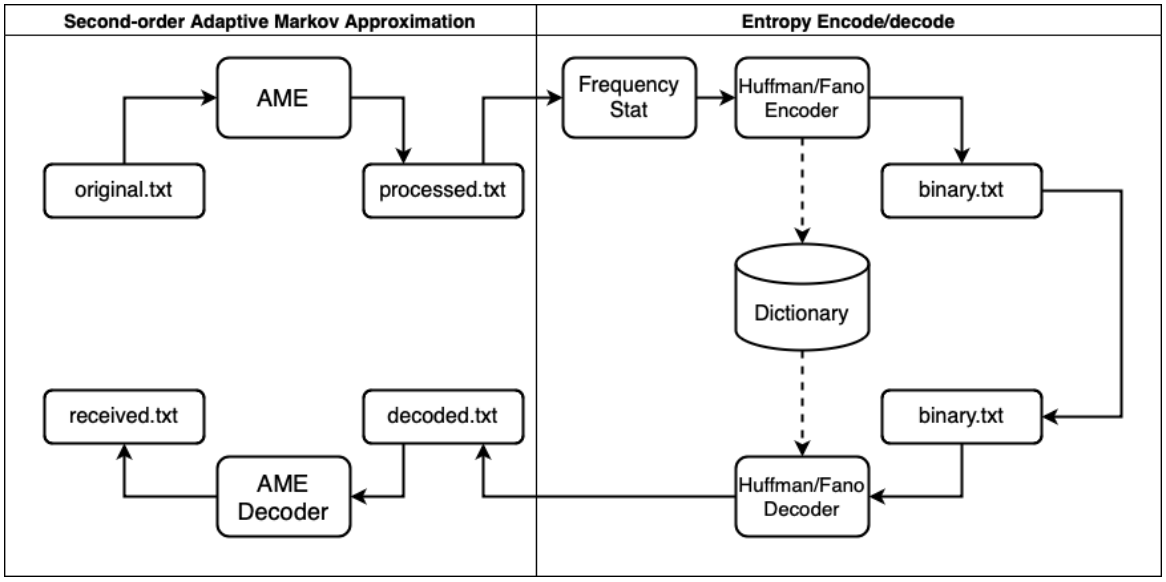

Figure 3: Flowchart of the experiment with AME

The only difference of this part is that we added a block called “Second-order Adaptive Markov Approximation” before applying the entropy encode/decode. Using the method in Sec-

[1]The dictionaries together with all codes can be found on https://github.com/Marcobisky/ame-entropy-source-coding

tion 3, we get the AME-encoded text (`processed.txt`). Part of the content of (`processed.txt`) is shown in Appendix I.

Then we apply the same procedure for `processed.txt` as we did for `original.txt`. Note this time the frequency of each character is different from the original text as shown in Table

Index	Symbol	Probability	Index	Symbol	Probability	Index	Symbol	Probability
1	(Space)	2.2477e-05	2	(CR)	0.0057	3	(LF)	0.0652
4	!	0.00022477	5	,	0.014	6	-	0.0011
7	.	0.0182	8	1	0.1721	9	2	0.0344
10	3	0.006	11	4	0.0057	12	5	0.0069
13	6	0.0017	14	7	0.00013486	15	8	2.2477e-05
16	:	4.4954e-05	17	;	0.00026972	18	?	0.0022
19	A	0.0015	20	B	0.0019	21	C	0.00060688
22	D	0.00038211	23	E	0.00042706	24	F	0.0008766
25	G	0.0012	26	H	0.0027	27	I	0.0028
28	J	0.0011	29	K	0.00022477	30	L	0.00060688
31	M	0.00056192	32	N	0.0013	33	O	0.00051697
34	P	0.00024725	35	R	0.0019	36	S	0.0016
37	T	0.0034	38	U	0.00015734	39	V	8.9908e-05
40	W	0.0036	41	Y	0.00067431	42	a	0.0571
43	b	0.0119	44	c	0.0148	45	d	0.0454
46	e	0.0484	47	f	0.0171	48	g	0.0176
49	h	0.0284	50	i	0.0459	51	j	0.00067431
52	k	0.0085	53	l	0.0319	54	m	0.0169
55	n	0.0259	56	o	0.0555	57	p	0.0087
58	q	0.00049449	59	r	0.0477	60	s	0.0501
61	t	0.039	62	u	0.0103	63	v	0.0055
64	w	0.0212	65	x	0.00031468	66	y	0.0153
67	z	0.00040459	68	’	0.0028	69	“	0.0056
70	”	0.0043	71	.	4.4954e-05			

After encoded into binary form, we send it to the receiver and decode it using the same dictionary to get the `decoded.txt`. However this is not the final result, we need to use the AME decoder to recover the original text. The AME decoder decodes the `decoded.txt` with no prior knowledge of the source. Since the markov chain is constructed in the same way, we are guaranteed to get the original text back.

6 Results and Discussion

6.1 Performance comparison between Huffman and Fano coding

The four main metrics of Huffman and Fano coding are shown in Table 3.

The four metrics are defined in Appendix J. Table 3 shown that Huffman coding outperforms Fano coding in all four metrics, which matches the fact that Huffman coding is the “optimal” coding strategy in the usual sense.

However, we noticed that the compression ratio achieved by Fano coding is greater than 1, which is unreasonable. With the following analysis, we claim it’s because this text is *unsuitable*

Table 3: Performance Metrics for Huffman and Fano Coding

Parameter	**Symbol**	**Huffman**	**Fano**
Average Code Length	$\bar{L}$	4.4769	8.6370
Code Rate	R	0.9897	0.5117
Efficiency	η	0.9897	0.5117
Compression Rate	ξ	0.5596	1.0796

for Fano Coding. Compared to other coding methods, the effectiveness of Fano Coding relies heavily on the uneven frequency distribution of symbols. Specifically, Fano Coding performs best when certain symbols appear very frequently while others are relatively rare. This frequency disparity allows shorter codes to be assigned to frequently occurring symbols, enabling compression.

However, in this text, the symbol frequency distribution is relatively uniform. The probability difference between frequently occurring symbols (such as spaces and letters) and rare symbols (such as punctuation marks) is not significant. In such cases of uniform symbol distribution, Fano Coding may fail to significantly reduce the file size. In fact, due to redundancy and long codewords, it may even increase the file size, leading to a compression ratio greater than 1. Moreover, we calculated the variance of the symbol frequencies in this text as follows:

```
>> var(probabilities)
ans =
    8.9864e-04
```

It is evident that the variance of the frequencies corresponding to these characters is very small, which further validates the point that the frequency distribution of symbols in this text is relatively uniform.

All above further highlights the superiority of Huffman Coding.

6.2 Performance analysis of AME coding

Since we pre-processed the `original.txt` before fed into the entropy coding, we cannot directly compute the four metrics for it. So we compared the length of `binary.txt` file generated in the two methods.

Let the length of the `binary_huffman.txt` and `binary_fano.txt` be L_{h} and L_{f}, respectively. Table 4 shows how using AME could save the length of the binary file.

Table 4: AME saves the length of binary file

	L_{h}	L_{f}
With AME	211066	212752
Without AME	219213	223058
Saved	96.28%	95.38%

7 Conclusion and Future Work

In this project, we successfully implemented the Huffman Coding technique using MATLAB, creating compressed files optimized for transmission and storage. Additionally, by comparing the results with Fano Coding, we explored the efficiency and optimization of Huffman Coding.

Furthermore, we introduced a new lossless coding scheme called 2nd-order Adaptive Markov Encoding (AME) coding, which serves as a pre-processor before applying entropy coding. It aims to uncover the structure of the source text and improve the compression ratio by extracting the "memoryless components" of the text.

Future work could involve using higher order Markov models or new mathematical models to capture more complex structures in the text, potentially furthermore improving the compression ratio.

References

[1] C. E. Shannon, "A Mathematical Theory of Communication".

[2] Advances in Communication and Computing Technologies (ICACACT 2014), Mumbai, India, 2014, pp. 1-6, doi: 10.1109/EIC.2015.7230711.

[3] N. Dhawale, "Implementation of Huffman algorithm and study for optimization," 2014 International Conference on Advances in Communication and Computing Technologies (ICACACT 2014), Mumbai, India, 2014, pp. 1-6, doi: 10.1109/EIC.2015.7230711.

[4] S. Congero and K. Zeger, "Competitive Advantage of Huffman and Shannon-Fano Codes," in IEEE Transactions on Information Theory, vol. 70, no. 11, pp. 7581-7598, Nov. 2024, doi: 10.1109/TIT.2024.3417010.

[5] T. Sharma et al., "A Survey on Machine Learning Techniques for Source Code Analysis," Sep. 13, 2022, arXiv: arXiv:2110.09610. doi: 10.48550/arXiv.2110.09610.

[6] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory, vol. 23, no. 3, pp. 337343, May 1977, doi: 10.1109/TIT.1977.1055714.

[7] A. D. Wyner and J. Ziv, The sliding-window Lempel-Ziv algorithm is asymptotically optimal, Proc. IEEE, vol. 82, no. 6, pp. 872877, Jun. 1994, doi: 10.1109/5.286191.

A AME encode/decode algorithm

Algorithm 1 Adaptive Markov Encoding

Require: Input file `source.txt`, Output file `markov_encoded.txt`
Ensure: Encoded text stored in `markov_encoded.txt`

Initialize an empty tree `tree`, list `encoded_output`, and counter `correct_predictions = 0`
Read `source_text` from `source.txt`
for `i = 1` to `length(source_text)` **do**
  `current_char = source_text[i]`
  if `i == 1` **then**
    Append `current_char` to `encoded_output`
    Add root-to-`current_char` transition to `tree`
  else
    Predict next character using `tree`: `prediction = predict_next(tree, prev_char)`
    if `prediction == current_char` **then**
      Increment `correct_predictions`
    else
      if `correct_predictions > 0` **then**
        Append `correct_predictions` to `encoded_output`
        Reset `correct_predictions = 0`
      end if
      Append `current_char` to `encoded_output`
    end if
    Add transition `prev_char` $\rightarrow$ `current_char` to `tree`
  end if
  `prev_char = current_char`
end for
if `correct_predictions > 0` **then**
  Append `correct_predictions` to `encoded_output`
end if
Write `encoded_output` to `markov_encoded.txt`

Algorithm 2 Adaptive Markov Decoding

Require: Input file `markov_encoded.txt`, Output file `markov_decoded.txt`
Ensure: Decoded text stored in `markov_decoded.txt`

```
Initialize an empty tree tree, list decoded_output
Read encoded_text from markov_encoded.txt
i = 1
while i ≤ length(encoded_text) do
  current_char = encoded_text[i]
  if i == 1 then
    Append current_char to decoded_output
    Add root-to-current_char transition to tree
  else
    prev_char = decoded_output[last]
    if current_char is a digit then
      Extract full number as repeat_count
      Predict next character using tree: prediction = predict_next(tree,
      prev_char)
      for j = 1 to repeat_count do
        Append prediction to decoded_output
        Add transition prev_char → prediction to tree
        prev_char = prediction
      end for
    else
      Append current_char to decoded_output
      Add transition prev_char → current_char to tree
    end if
  end if
  i = i + 1
end while
Write decoded_output to markov_decoded.txt
```

B Python code for AME encoding

Listing 1: AME encoder

```python
# adaptive_markov_encode.py
import json

def add_to_tree(tree, from_char, to_char):
    """Add a transition to the tree."""
    if from_char not in tree:
        tree[from_char] = {"transitions": {}, "highestFrequency": 0, "probableNextChar": ""}

    transitions = tree[from_char]["transitions"]
    if to_char in transitions:
        transitions[to_char] += 1
    else:
        transitions[to_char] = 1

    if transitions[to_char] >= tree[from_char]["highestFrequency"]:
        tree[from_char]["highestFrequency"] = transitions[to_char]
        tree[from_char]["probableNextChar"] = to_char

def predict_next(tree, current_char):
    """Predict the next character based on tree."""
    if current_char not in tree:
        return ""
    return tree[current_char]["probableNextChar"]

def encode(source_file, output_file):
    """Encodes a file using second-order Markov approximation."""
    with open(source_file, "rb") as f:
        source_text = f.read().decode()

    tree = {}
    encoded_output = []
    correct_predictions = 0

    for i, current_char in enumerate(source_text):
        if i == 0:
            add_to_tree(tree, "", current_char)
            encoded_output.append(current_char)
        else:
            prev_char = source_text[i - 1]
            prediction = predict_next(tree, prev_char)

            if prediction == current_char:
                correct_predictions += 1
            else:
                if correct_predictions > 0:
                    encoded_output.append(str(correct_predictions))
```

```
                    correct_predictions = 0
                encoded_output.append(current_char)
            add_to_tree(tree, prev_char, current_char)

    if correct_predictions > 0:
        encoded_output.append(str(correct_predictions))

    with open(output_file, "wb") as f:
        f.write("".join(encoded_output).encode())

if __name__ == "__main__":
    encode("original.txt", "processed.txt")
```

C Python code for AME decoding

Listing 2: AME decoder

```python
# adaptive_markov_decode.py
import json

def add_to_tree(tree, from_char, to_char):
    """Add a transition to the tree."""
    if from_char not in tree:
        tree[from_char] = {"transitions": {}, "highestFrequency": 0, "probableNextChar": ""}

    transitions = tree[from_char]["transitions"]
    if to_char in transitions:
        transitions[to_char] += 1
    else:
        transitions[to_char] = 1

    if transitions[to_char] >= tree[from_char]["highestFrequency"]:
        tree[from_char]["highestFrequency"] = transitions[to_char]
        tree[from_char]["probableNextChar"] = to_char

def predict_next(tree, current_char):
    """Predict the next character based on tree."""
    if current_char not in tree:
        return ""
    return tree[current_char]["probableNextChar"]

def decode(encoded_file, decoded_file):
    """Decodes a file using second-order Markov approximation."""
    with open(encoded_file, "rb") as f:
        encoded_text = f.read().decode()

    tree = {}
    decoded_output = []
    i = 0

    while i < len(encoded_text):
        current_char = encoded_text[i]

        if i == 0:
            # First character
            decoded_output.append(current_char)
            add_to_tree(tree, "", current_char)
        else:
            prev_char = decoded_output[-1]
            if current_char.isdigit():
                num_str = ""
                while i < len(encoded_text) and encoded_text[i].isdigit():
                    num_str += encoded_text[i]
```

```
                    i += 1
                i -= 1
                repeat_count = int(num_str)
                for _ in range(repeat_count):
                    prediction = predict_next(tree, prev_char)
                    if not prediction:
                        raise ValueError(f"Decoding error: No valid
    prediction for {prev_char}.")
                    add_to_tree(tree, prev_char, prediction)
                    decoded_output.append(prediction)
                    prev_char = prediction
            else:
                decoded_output.append(current_char)
                add_to_tree(tree, prev_char, current_char)

        i += 1

    with open(decoded_file, "wb") as f:
        f.write("".join(decoded_output).encode())

if __name__ == "__main__":
    decode("processed.txt", "received.txt")
```

D Huffman Tree Generated for `original.txt`

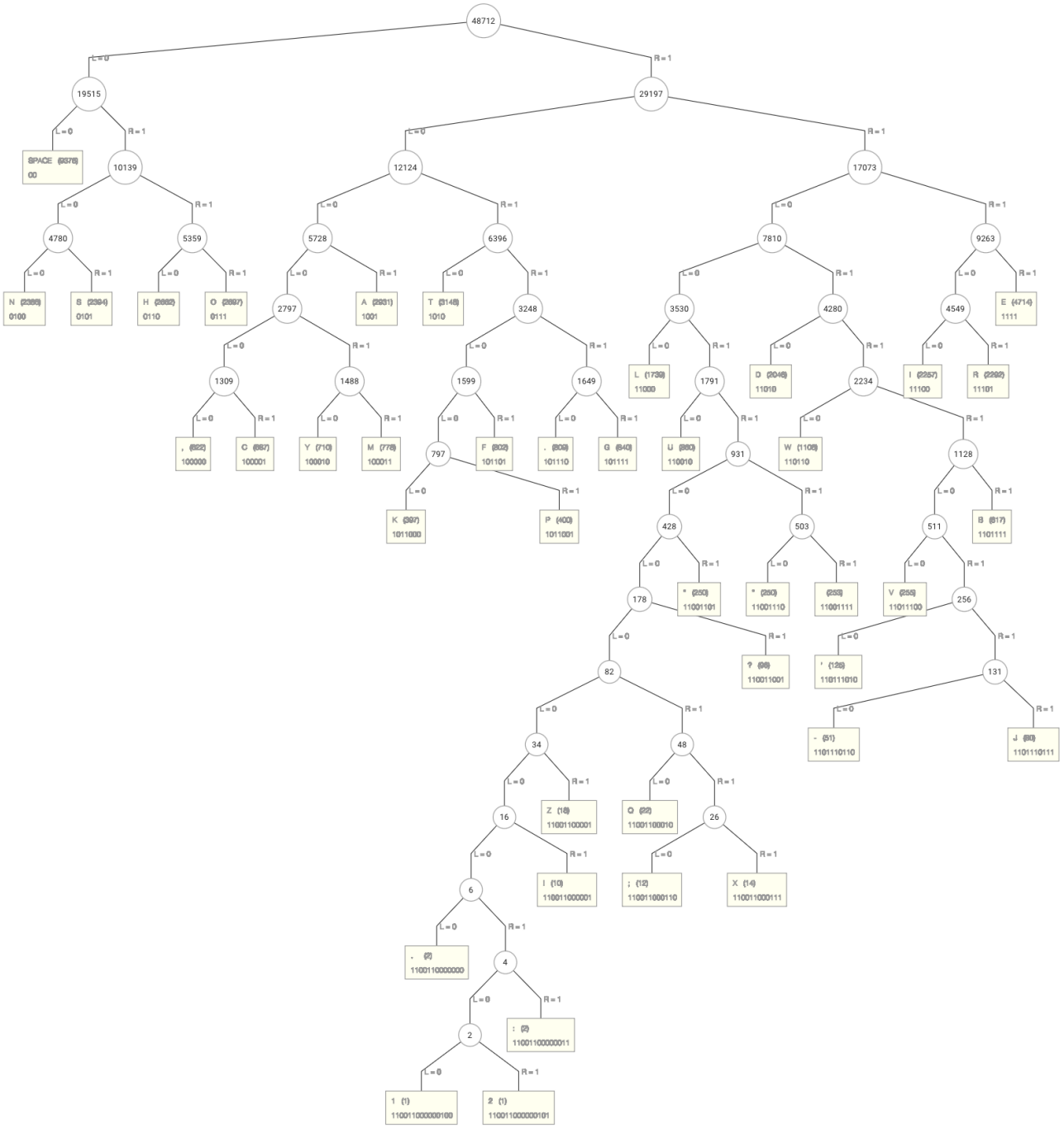

Figure 4: Huffman tree visualization

E MATLAB code for huffman encode/decode procedure

Listing 3: Huffman encode procedure (without AME example)

```
function [encodedMessage, dict] = huffman_encode(text)
    % compute frequency
    symbols = unique(text);
    counts = histc(text, symbols);

    %change symbols into cell
    symbols = num2cell(symbols);

    % calculate probabiliyies
    probabilities = counts / sum(counts);


    dict = huffmandict(symbols, probabilities);

    % encode
    encodedMessage = huffmanenco(num2cell(text), dict);
end
```

Listing 4: Huffman decode procedure (without AME example)

```
function decodedMessage = huffman_decode(encodedMessage, dict)
    % Decode encoded information using Huffman dictionary
    decodedMessage = huffmandeco(encodedMessage, dict);

    % Converts the decoded cell array back to a character array
    decodedMessage = cell2mat(decodedMessage);
end
```

Listing 5: Main function to perform Huffman encode/decode

```
% 1.initialise four different paths
inputFilePath = 'original.txt'; % the original text
encodedFilePath = 'binary_huffman.txt'; % binary-encoded text file
dictFilePath = 'Huffman_dictionary.txt'; % dictionary: Huffman code for each
   character
decodeFilePath = 'received_huffman.txt'; % decoded text file

% 2.read input text
fileID = fopen(inputFilePath, 'r');
text = fscanf(fileID, '%c');
fclose(fileID);

% 3.encode that original text using Huffman decoding and put that in a .txt
   file
[encoded_text, dict] = huffman_encode(text);
fileID = fopen(encodedFilePath, 'w');
fprintf(fileID, '%d', encoded_text);
fclose(fileID);
```

```
17
18 % 4.put the dictionary in a .txt file and display that
19 fileID = fopen(dictFilePath, 'w');
20 for i = 1:length(dict)
21     fprintf(fileID, '%s: %s\n', num2str(cell2mat(dict(i, 1))), num2str(
       cell2mat(dict(i, 2))));
22 end
23 fclose(fileID);
24
25 % 5.decode the encoded text and put that in a .txt file using Huffman decoding
26 decodedMessage = huffman_decode(encoded_text, dict);
27 fileID = fopen(decodeFilePath, 'w');
28 fprintf(fileID, '%s', decodedMessage);
29 fclose(fileID);
30
31 % 6.verify if decoding was successful
32 if isequal(text, decodedMessage)
33     disp('Decoding is successful!');
34 else
35     disp('Decoding failed.');
36 end
37
38 % 7.calculate some relevant parameters and display them
39 [avgCodeLength, rate, efficiency,zip_rate] = calculate_encoding_metrics(text,
       dict);
40 disp('average code length:');disp(avgCodeLength);
41 disp('code rate:');disp(rate);
42 disp('efficiency:');disp(efficiency);
43 disp('zip_rate:');disp(zip_rate);
```

F MATLAB code for huffman encode/decode procedure

Listing 6: Fano encode procedure (without AME example)

```
function [encodedMessage, dict] = fano_encode(text)
    % Compute symbol frequencies
    symbols = unique(text); % Unique characters, including spaces
    counts = histc(text, symbols); % Count occurrences

    % Convert symbols into cell format for compatibility
    symbols = num2cell(symbols);

    % Calculate probabilities
    probabilities = counts / sum(counts);

    % Sort symbols by probabilities in descending order
    [probabilities, idx] = sort(probabilities, 'descend');
    symbols = symbols(idx);

    % Generate Fano code dictionary
    dict = fano_create_dict(symbols, probabilities);

    % Encode the text
    encodedMessage = '';
    for i = 1:length(text)
        % Match each character to its code in the dictionary
        symbol = text(i);
        code = dict{ismember(dict(:,1), {symbol}), 2};
        encodedMessage = strcat(encodedMessage, code);
    end
end

function dict = fano_create_dict(symbols, probabilities)
    % Initialize dictionary
    dict = cell(length(symbols), 2);
    dict(:, 1) = symbols; % Add symbols to the dictionary

    % Recursive function to generate Fano codes
    function generate_code(subset, code)
        if numel(subset) == 1
            % Assign code to the only symbol left
            dict{ismember(dict(:,1), subset{1}), 2} = code;
            return;
        end

        % Find the partition point
        cumulative = cumsum(probabilities(ismember(symbols, subset)));
        total = cumulative(end);
        partition = find(cumulative >= total / 2, 1);

        % Split symbols into two groups and assign 0/1
```

```
        generate_code(subset(1:partition), [code, '0']);
        generate_code(subset(partition+1:end), [code, '1']);
    end

    % Start the recursive encoding
    generate_code(symbols, '');
end
```

Listing 7: Fano decode procedure (without AME example)

```
function decodedMessage = fano_decode(encodedMessage, dict)
    % Decode the encoded binary string using the Fano dictionary

    % Convert the dictionary into a map for fast lookup
    codeMap = containers.Map(dict(:,2), dict(:,1));

    % Initialize decoding variables
    decodedMessage = '';
    buffer = '';

    % Iterate through the encoded message to decode
    for i = 1:length(encodedMessage)
        buffer = strcat(buffer, encodedMessage(i)); % Append bit to buffer

        if isKey(codeMap, buffer)
            % If the buffer matches a code in the dictionary
            decodedChar = codeMap(buffer);
            decodedMessage = strcat(decodedMessage, decodedChar);
            buffer = ''; % Reset the buffer
        end
    end

    % Ensure the decoded message matches the original text
    if ~isempty(buffer)
        error('Decoding error: incomplete or invalid encoding.');
    end
end
```

Listing 8: Main function to perform Fano encode/decode

```
% 1.initialise four different paths
inputFilePath = 'original.txt';% the original text
encodedFilePath = 'binary_fano.txt';% binary-encoded text file
dictFilePath = 'Fano_dictionary.txt';% dictionary: Huffman code for each
   character
decodedFilePath = 'received_fano.txt';% decoded text file

% 2.read input text
fileID = fopen(inputFilePath, 'r');
text = fread(fileID, '*char')'; % Read all characters, preserving spaces and
   line endings
fclose(fileID);
```

```
% replace CRLF with a unique marker to ensure consistent handling
crlfMarker = '\r\n'; % Marker for CRLF during encoding/decoding
text = strrep(text, char([13 10]), crlfMarker); % Replace \r\n with marker

% 3.encode that original text using Fano decoding and put that in a .txt file
[encodedMessage, dict] = fano_encode(text);
% Ensure spaces are correctly encoded and stored in the dictionary
dict{strcmp(dict(:, 1), ' '), 1} = '_'; % Map space to underscore
fileID = fopen(encodedFilePath, 'w');
fprintf(fileID, '%s', encodedMessage); % Write as binary string
fclose(fileID);

% 4.put the Fano dictionary in a .txt file with CRLF format
fileID = fopen(dictFilePath, 'w');
for i = 1:size(dict, 1)
    line = sprintf('%s: %s\r\n', char(dict{i, 1}), dict{i, 2}); % Use \r\n for
    CRLF
    fwrite(fileID, line, 'char'); % Write each line explicitly with CRLF
end
fclose(fileID);

% 5.decode the encoded text and put that in a .txt file using Fano decoding
decodedMessage = fano_decode(encodedMessage, dict);
% ensure underscores are converted back to spaces
decodedMessage = strrep(decodedMessage, '_', ' ');
% restore original CRLF line endings (without markers in the output file)
decodedMessage = strrep(decodedMessage, crlfMarker, char([13 10])); % Replace
   marker with \r\n
% write the decoded message to a file
fileID = fopen(decodedFilePath, 'w');
fwrite(fileID, decodedMessage, 'char'); % Use fwrite to ensure exact output
   including spaces and line endings
fclose(fileID);

% 6.verify if decoding was successful
if isequal(strrep(text, crlfMarker, char([13 10])), decodedMessage)
    disp('Decoding is successful!');
else
    disp('Decoding failed.');
end

% 7.calculate some relevant parameters and display them
[avgCodeLength, rate, efficiency,zip_rate] = calculate_encoding_metrics(text,
   dict);
disp('average code length:');disp(avgCodeLength);
disp('code rate:');disp(rate);
disp('efficiency:');disp(efficiency);
disp('zip_rate:');disp(zip_rate);
```

G Part of the `original.txt`

```
PROLOGUE
  We should start back, Gared urged as the woods began to grow dark around them. The wildlings are dead.
  Do the dead frighten you? Ser Waymar Royce asked with just the hint of a smile.
  Gared did not rise to the bait. He was an old man, past fifty, and he had seen the lordlings come and go. Dead is dead, he said. We have no business with the dead.
  Are they dead? Royce asked softly. What proof have we? ...
```

H Part of the `binary_huffman.txt`

```
1101011010110110011100101011010101010010010100101110101010100100100100111110101
1001111001100101011001011001111111001110100001100010101010000011110100000011011
0001001100111000110110110010100011110010101011001001100010111011010010010001111
0001100000011011001111000110101000000001101101110001011011010100000110011101000
0100000110000101100101000000010100011110111101101000110101000000111000001110110
0110011100011010110011011100011100000111110110011011011010100000010111010011111
0010111101011011101000001100111000100010000011001000001001011010100000101101111
0001100001100110000001110011001001100101110001010110010110011111110010111110011
1110011000110110101000001100110000001110011011010101000110010001010010100110000
0010111110010010000011110101101000010111011100111101000000011111001110101111000
1000101110111000111110011100110001001001001010000110111000100101100000000011011
```

I Part of the `processed.txt`

```
PROLOGUE
  1We should start back, Gared1urge2as the woods1began to grow da1k a1ound1th1m. Th2wil1lings1a1e1dead.
2 1Do t3dead1frighten you? Ser Wayma1 Royce1asked1w1t1 just t3hint of a smile.
2 1G2ed1did1not rise1to 4bait. He1was1an1ol2man, past fifty,1 and1h2had1seen1t3lordlings1come1and1go.11ead1is1dead,1h2said.1 We1have1no1bu1i1ess1w1t1 t3dead.
2  Ar5y dead?2R1yce1ask1d1sof1ly.1What proof hav2we?1
```

J Metrics Definitions

The four main metrics of entropy coding include average code length $\bar{L}$, code rate R, efficiency η, and compression ratio ξ. They are defined as follows:

Let L_i and p_i be the length and probability of the i-th symbol, respectively. The average code length $\bar{L}$ is defined as:

$$\bar{L} = \sum_{i=1}^{n} L_i \cdot p_i. \tag{1}$$

Let the source entropy be $H(S)$. The code rate R is defined as:

$$R = \frac{H(S)}{\bar{L}}. \tag{2}$$

Let the code be n-ary. The efficiency η is defined as:

$$\eta = \frac{R}{\log_2(n)} = \frac{H(S)}{\bar{L} \cdot \log_2(n)}. \tag{3}$$

The compression ratio ξ is defined as (w.r.t. ACSII encoding):

$$\xi = \frac{\bar{L}}{8}. \tag{4}$$